Device mapper io stacks are configured in bottom-up fashion. Target driver devices are stacked by referencing a lower level mapped device as a target device of a higher level mapped device. Since a single mapped device may map to one or more target devices, each of which may themselves be a mapped device, a device mapper io stack may be more accurately viewed as an inverted device tree with a single mapped device as the top or root node of the inverted tree. The leaf nodes of the tree are the only target devices which are not device mapper managed devices. The root node is only a mapped device. Every non-root, nonleaf node is both a mapped and target device.
The minimum device tree consists of a single mapped device and a single target device. A device tree need not be balanced as there may be device branches which are deeper than others. The depth of the tree may be viewed as the tree branch which has the maximum number of transitions from the root mapped device to leaf node target device. There are no design limits on either the depth or breadth of a device tree. Although each target device at each level of a device mapper tree is visible and accessible outside the scope of the device mapper framework, concurrent open of a target device for other purposes requiring its exclusive use such as is required for partition management and file system mounting is prohibited. Target devices are exclusively recognized or claimed by a mapped device by being referenced as a target of a mapped device. That is, a target device may only be used as a target of a single mapped device. This restriction prohibits both the inclusion of the same target device within multiple device trees and multiple references to the same target device within the same device tree, that is, loops within a device tree are not allowed.
It is strictly the responsibility of user space components associated with each target driver
To

- discover the set of associated target devices associated with each mapped device managed by that driver
- create the mapping tables containing this configuration information
- pass the mapping table information into the kernel
- possibly save this mapping information in persistent storage for later retrieval.

The multipath path configurator fulfills this role for the multipathing target driver. The lvm(8), dmraid(8), and dmsetup(8) commands perform these tasks for the logical volume management, software raid, and the device encryption target drivers respectively.
While the device mapper registers with the kernel as a block device driver, target drivers in turn register callbacks with the device mapper for initializing and terminating target device metadata; suspending and resuming io on a mapped device; filtering io dispatch and io completion; and retrieving mapped

device configuration and status information. The device mapper also provides key services, (e.g., io suspension/ resumption, bio cloning, and the propagation of io resource restrictions), for use by all target drivers to facilitate the flow of io dispatch and io completion events through the device mapper framework.
The device mapper framework is itself a component driver within the outermost generic_make_request framework for block devices.
The generic_make_request framework also provides for stacking block device filter drivers. Therefore, given this architecture, it should be at least architecturally possible to stack device mapper drivers both above and below multidisk drivers for the same target device. The device mapper processes all read and write block io requests which pass through the block io subsystem's generic_make_request and/or submit_bio interfaces and is directed to a mapped device. Architectural symmetry is achieved for io dispatch and io completion handling since io completion handling within the device mapper framework is done in the inverse order of io dispatch. All read/write bios are treated as asynchronous io within all portions of the block io subsystem. This design results in separate, asynchronous and inversely ordered code paths through both the generic_make_request and the device mapper frameworks for both io dispatch and completion processing. A major impact of this design is that it is not necessary to process either an io dispatch or completion either immediately or in the same context in which they are first seen.
Bio movement through a device mapper device tree may involve fan-out on bio dispatch and fan-in on bio completion. As a bio is dispatched down the device tree at each mapped device, one or more cloned copies of the bio are created and sent to target devices. The same process is repeated at each level of the device tree where a target device is also a mapped device. Therefore, assuming a very wide and deep device tree, a single bio dispatched to a mapped device can branch out to spawn a practically unbounded number of bios to be sent to a practically unbounded number of target devices. Since bios are potentially coalesced at the device at the bottom of the generic_make_request framework, the io request(s) actually queued to one or more target devices at the bottom may bear little relationship to the single bio initially sent to a mapped device at the top. For bio completion, at each level of the device tree, the target driver managing the set of target devices at that level consumes the completion for each bio dispatched to one of its devices, and passes up a single bio completion for the single bio dispatched to the mapped device.
This process repeats until the original bio submitted to the root mapped device is completed. The device mapper dispatches bios recursively from top (root node) to bottom (leaf node) through the tree of device mapper mapped and target devices in process context. Each level of recursion moves down one level of the device tree from the root mapped device to one or more leaf target nodes. At each level, the device mapper clones a single bio to one or more bios

depending on target mapping information previously pushed into the kernel for each mapped device in the io stack since a bio is not permitted to span multiple map targets/segments. Also at each level, each cloned bio is passed to the map callout of the target driver managing a mapped device. The target driver has the option of

1. queuing the io internal to that driver to be serviced at a later time by that driver,
2. redirecting the io to one or more different 0target devices and possibly a different sector
on each of those target devices, or
3. returning an error status for the bio to the device mapper.

Both the first or third options stop the recursion through the device tree and the generic_make_request framework for that matter.
Otherwise, a bio being directed to the first target device which is not managed by the device mapper causes the bio to exit the device mapper framework, although the bio continues recursing through the generic_make_request framework until the bottom device is reached.
The device mapper processes bio completions recursively from a leaf device to the root mapped device in soft interrupt context. At each level in a device tree, bio completions are filtered by the device mapper as a result of redirecting the bio completion callback at that level during bio dispatch. The device mapper callout to the target driver responsible for servicing a mapped device is enabled by associating a target_io structure with the bi_private field of a bio, also during bio dispatch. In this fashion, each bio completion is serviced by the target driver which dispatched the bio.
The device mapper supports a variety of push/pull interfaces to enhance communication across the system call boundary. Each of these interfaces is accessed from user space via the device mapper library which currently issues ioctls to the device mapper character interface. The occurrence of target driver derived io related events can be passed to user space via the device mapper event mechanism. Target driver specific map contents and mapped device status can be pulled from the kernel using device mapper messages. Typed messages and status information are encoded as ASCII strings and decoded back to their original form according dictated by their type.