USB4 and Thunderbolt

USB4 is the public specification based on Thunderbolt 3 protocol with some differences at the register level among other things. Connection manager is an entity running on the host router (host controller) responsible for enumerating routers and establishing tunnels. A connection manager can be implemented either in firmware or software. Typically PCs come with a firmware connection manager for Thunderbolt 3 and early USB4 capable systems. Apple systems on the other hand use software connection manager and the later USB4 compliant devices follow the suit.

The Linux Thunderbolt driver supports both and can detect at runtime which connection manager implementation is to be used. To be on the safe side the software connection manager in Linux also advertises security level `user` which means PCIe tunneling is disabled by default. The documentation below applies to both implementations with the exception that the software connection manager only supports `user` security level and is expected to be accompanied with an IOMMU based DMA protection.

Security levels and how to use them

The interface presented here is not meant for end users. Instead there should be a userspace tool that handles all the low-level details, keeps a database of the authorized devices and prompts users for new connections.

More details about the sysfs interface for Thunderbolt devices can be found in `Documentation/ABI/testing/sysfs-bus-thunderbolt`.

Those users who just want to connect any device without any sort of manual work can add following line to `/etc/udev/rules.d/99-local.rules`:

```
ACTION=="add", SUBSYSTEM=="thunderbolt", ATTR{authorized}=="0", ATTR{authorized}="1"
```

This will authorize all devices automatically when they appear. However, keep in mind that this bypasses the security levels and makes the system vulnerable to DMA attacks.

Starting with Intel Falcon Ridge Thunderbolt controller there are 4 security levels available. Intel Titan Ridge added one more security level (usbonly). The reason for these is the fact that the connected devices can be DMA masters and thus read contents of the host memory without CPU and OS knowing about it. There are ways to prevent this by setting up an IOMMU but it is not always available for various reasons.

Some USB4 systems have a BIOS setting to disable PCIe tunneling. This is treated as another security level (nopcie).

The security levels are as follows:

none
: All devices are automatically connected by the firmware. No user approval is needed. In BIOS settings this is typically called *Legacy mode*.

user
: User is asked whether the device is allowed to be connected. Based on the device identification information available through `/sys/bus/thunderbolt/devices`, the user then can make the decision. In BIOS settings this is typically called *Unique ID*.

secure
: User is asked whether the device is allowed to be connected. In addition to UUID the device (if it supports secure connect) is sent a challenge that should match the expected one based on a random key written to the `key` sysfs attribute. In BIOS settings this is typically called *One time saved key*.

dponly
: The firmware automatically creates tunnels for Display Port and USB. No PCIe tunneling is done. In BIOS settings this is typically called *Display Port Only*.

usbonly
: The firmware automatically creates tunnels for the USB controller and Display Port in a dock. All PCIe links downstream of the dock are removed.

nopcie
: PCIe tunneling is disabled/forbidden from the BIOS. Available in some USB4 systems.

The current security level can be read from `/sys/bus/thunderbolt/devices/domainX/security` where `domainX` is the Thunderbolt domain the host controller manages. There is typically one domain per Thunderbolt host controller.

If the security level reads as `user` or `secure` the connected device must be authorized by the user before PCIe tunnels are created (e.g the PCIe device appears).

Each Thunderbolt device plugged in will appear in sysfs under `/sys/bus/thunderbolt/devices`. The device directory carries information that can be used to identify the particular device, including its name and UUID.

Authorizing devices when security level is `user` or `secure`

When a device is plugged in it will appear in sysfs as follows:

```
/sys/bus/thunderbolt/devices/0-1/authorized   - 0
/sys/bus/thunderbolt/devices/0-1/device       - 0x8004
/sys/bus/thunderbolt/devices/0-1/device_name  - Thunderbolt to FireWire Adapter
/sys/bus/thunderbolt/devices/0-1/vendor       - 0x1
/sys/bus/thunderbolt/devices/0-1/vendor_name  - Apple, Inc.
/sys/bus/thunderbolt/devices/0-1/unique_id    - e0376f00-0300-0100-ffff-ffffffffffff
```

The `authorized` attribute reads 0 which means no PCIe tunnels are created yet. The user can authorize the device by simply entering:

```
# echo 1 > /sys/bus/thunderbolt/devices/0-1/authorized
```

This will create the PCIe tunnels and the device is now connected.

If the device supports secure connect, and the domain security level is set to `secure`, it has an additional attribute `key` which can hold a random 32-byte value used for authorization and challenging the device in future connects:

```
/sys/bus/thunderbolt/devices/0-3/authorized   - 0
/sys/bus/thunderbolt/devices/0-3/device       - 0x305
/sys/bus/thunderbolt/devices/0-3/device_name  - AKiTiO Thunder3 PCIe Box
/sys/bus/thunderbolt/devices/0-3/key          -
/sys/bus/thunderbolt/devices/0-3/vendor       - 0x41
/sys/bus/thunderbolt/devices/0-3/vendor_name  - inXtron
/sys/bus/thunderbolt/devices/0-3/unique_id    - dc010000-0000-8508-a22d-32ca6421cb16
```

Notice the key is empty by default.

If the user does not want to use secure connect they can just `echo 1` to the `authorized` attribute and the PCIe tunnels will be created in the same way as in the `user` security level.

If the user wants to use secure connect, the first time the device is plugged a key needs to be created and sent to the device:

```
# key=$(openssl rand -hex 32)
# echo $key > /sys/bus/thunderbolt/devices/0-3/key
# echo 1 > /sys/bus/thunderbolt/devices/0-3/authorized
```

Now the device is connected (PCIe tunnels are created) and in addition the key is stored on the device NVM.

Next time the device is plugged in the user can verify (challenge) the device using the same key:

```
# echo $key > /sys/bus/thunderbolt/devices/0-3/key
# echo 2 > /sys/bus/thunderbolt/devices/0-3/authorized
```

If the challenge the device returns back matches the one we expect based on the key, the device is connected and the PCIe tunnels are created. However, if the challenge fails no tunnels are created and error is returned to the user.

If the user still wants to connect the device they can either approve the device without a key or write a new key and write 1 to the `authorized` file to get the new key stored on the device NVM.

De-authorizing devices

It is possible to de-authorize devices by writing `0` to their `authorized` attribute. This requires support from the connection manager implementation and can be checked by reading domain `deauthorization` attribute. If it reads `1` then the feature is supported.

When a device is de-authorized the PCIe tunnel from the parent device PCIe downstream (or root) port to the device PCIe upstream port is torn down. This is essentially the same thing as PCIe hot-remove and the PCIe toplogy in question will not be accessible anymore until the device is authorized again. If there is storage such as NVMe or similar involved, there is a risk for data loss if the filesystem on that storage is not properly shut down. You have been warned!

DMA protection utilizing IOMMU

Recent systems from 2018 and forward with Thunderbolt ports may natively support IOMMU. This means that Thunderbolt security is handled by an IOMMU so connected devices cannot access memory regions outside of what is allocated for them by drivers. When Linux is running on such system it automatically enables IOMMU if not enabled by the user already. These systems can be identified by reading `1` from `/sys/bus/thunderbolt/devices/domainX/iommu_dma_protection` attribute.

The driver does not do anything special in this case but because DMA protection is handled by the IOMMU, security levels (if set) are redundant. For this reason some systems ship with security level set to `none`. Other systems have security level set to `user` in order to support downgrade to older OS, so users who want to automatically authorize devices when IOMMU DMA protection is enabled can use the following `udev` rule:

```
ACTION=="add", SUBSYSTEM=="thunderbolt", ATTRS{iommu_dma_protection}=="1", ATTR{authorized}=="0", ATTR{authori
```

Upgrading NVM on Thunderbolt device, host or retimer

Since most of the functionality is handled in firmware running on a host controller or a device, it is important that the firmware can be upgraded to the latest where possible bugs in it have been fixed. Typically OEMs provide this firmware from their support site.

There is also a central site which has links where to download firmware for some machines:

Thunderbolt Updates

Before you upgrade firmware on a device, host or retimer, please make sure it is a suitable upgrade. Failing to do that may render the device in a state where it cannot be used properly anymore without special tools!

Host NVM upgrade on Apple Macs is not supported.

Once the NVM image has been downloaded, you need to plug in a Thunderbolt device so that the host controller appears. It does not matter which device is connected (unless you are upgrading NVM on a device - then you need to connect that particular device).

Note an OEM-specific method to power the controller up ("force power") may be available for your system in which case there is no need to plug in a Thunderbolt device.

After that we can write the firmware to the non-active parts of the NVM of the host or device. As an example here is how Intel NUC6i7KYK (Skull Canyon) Thunderbolt controller NVM is upgraded:

```
# dd if=KYK_TBT_FW_0018.bin of=/sys/bus/thunderbolt/devices/0-0/nvm_non_active0/nvmem
```

Once the operation completes we can trigger NVM authentication and upgrade process as follows:

```
# echo 1 > /sys/bus/thunderbolt/devices/0-0/nvm_authenticate
```

If no errors are returned, the host controller shortly disappears. Once it comes back the driver notices it and initiates a full power cycle. After a while the host controller appears again and this time it should be fully functional.

We can verify that the new NVM firmware is active by running the following commands:

```
# cat /sys/bus/thunderbolt/devices/0-0/nvm_authenticate
0x0
# cat /sys/bus/thunderbolt/devices/0-0/nvm_version
18.0
```

If `nvm_authenticate` contains anything other than 0x0 it is the error code from the last authentication cycle, which means the authentication of the NVM image failed.

Note names of the NVMem devices `nvm_activeN` and `nvm_non_activeN` depend on the order they are registered in the NVMem subsystem. N in the name is the identifier added by the NVMem subsystem.

Upgrading on-board retimer NVM when there is no cable connected

If the platform supports, it may be possible to upgrade the retimer NVM firmware even when there is nothing connected to the USB4 ports. When this is the case the `usb4_portX` devices have two special attributes: `offline` and `rescan`. The way to upgrade the firmware is to first put the USB4 port into offline mode:

```
# echo 1 > /sys/bus/thunderbolt/devices/0-0/usb4_port1/offline
```

This step makes sure the port does not respond to any hotplug events, and also ensures the retimers are powered on. The next step is to scan for the retimers:

```
# echo 1 > /sys/bus/thunderbolt/devices/0-0/usb4_port1/rescan
```

This enumerates and adds the on-board retimers. Now retimer NVM can be upgraded in the same way than with cable connected (see previous section). However, the retimer is not disconnected as we are offline mode) so after writing `1` to `nvm_authenticate` one should wait for 5 or more seconds before running rescan again:

```
# echo 1 > /sys/bus/thunderbolt/devices/0-0/usb4_port1/rescan
```

This point if everything went fine, the port can be put back to functional state again:

```
# echo 0 > /sys/bus/thunderbolt/devices/0-0/usb4_port1/offline
```

Upgrading NVM when host controller is in safe mode

If the existing NVM is not properly authenticated (or is missing) the host controller goes into safe mode which means that the only available functionality is flashing a new NVM image. When in this mode, reading `nvm_version` fails with `ENODATA` and the device identification information is missing.

To recover from this mode, one needs to flash a valid NVM image to the host controller in the same way it is done in the previous chapter.

Networking over Thunderbolt cable

Thunderbolt technology allows software communication between two hosts connected by a Thunderbolt cable.

It is possible to tunnel any kind of traffic over a Thunderbolt link but currently we only support Apple ThunderboltIP protocol.

If the other host is running Windows or macOS, the only thing you need to do is to connect a Thunderbolt cable between the two hosts; the `thunderbolt-net` driver is loaded automatically. If the other host is also Linux you should load `thunderbolt-net` manually on one host (it does not matter which one):

```
# modprobe thunderbolt-net
```

This triggers module load on the other host automatically. If the driver is built-in to the kernel image, there is no need to do anything.

The driver will create one virtual ethernet interface per Thunderbolt port which are named like `thunderbolt0` and so on. From this point you can either use standard userspace tools like `ifconfig` to configure the interface or let your GUI handle it automatically.

Forcing power

Many OEMs include a method that can be used to force the power of a Thunderbolt controller to an "On" state even if nothing is connected. If supported by your machine this will be exposed by the WMI bus with a sysfs attribute called "force_power".

For example the intel-wmi-thunderbolt driver exposes this attribute in:

> /sys/bus/wmi/devices/86CCFD48-205E-4A77-9C48-2021CBEDE341/force_power
>
> To force the power to on, write 1 to this attribute file. To disable force power, write 0 to this attribute file.

Note: it's currently not possible to query the force power state of a platform.