We will have two drivers: -

1. Root-complex.

This is a standard PCIe driver, so you'll just follow convention there.

1. End-point driver.

This driver needs to use the PCIe bus, but its not responsible for the PCIe bus in the way a root-complex is. The driver needs to know what the root-complex is interrupting it for, eg., "transmitter empty" (I've read your last message) or "receiver ready" (there is a message from me, waiting for you).

So you need at least two unique interrupts or messages from the root-complex to the end-point.

I am happy to inform you that I finally found a way to register for the interrupts from RC to EP. Now I have made a simple root and end point network driver for two MPC8640 nodes that are now up and running and I could successfully ping across them.

The basic flow is as follows.

**Root Complex Driver: -**

1. It discovers the EP processor node and gets its base addresses.(BAR 1 and BAR 2)

2. It sets a single inbound window mapping a portion of its RAM to PCI space.(This is to allow inbound memory writes from EP).

3. It enables the MSI interrupt for the EP and registers an interrupt handler for the same.(To receive interrupts from EP. Note this is conventional PCI method)

4. On receiving a transmit request from kernel it initiates a DMA memory copy of the packet (in the socket buffer) to the EP memory through BAR 1. After DMA finishes it sends an interrupt to EP by writing to its msi register mapped in BAR2.

5. On reception of a packet (from EP) the msi interrupt handler is called and it copies the packet in RAM to a socket buffer and passes it to the kernel.

**End Point Driver: -**

1. It sets up the internal msi interrupt structure and registers an interrupt handler. (To receive interrupts from RC. Note this is not done by default in kernel as it is a slave and thus is added in the driver.)

2. It sets two inbound windows

i) BAR1 maps to RAM area.(To allow inbound memory write from RC)

ii) BAR2 is mapped to PIC register area.(To allow inbound message Interrupt register write from RC)

3. It sets up one outbound window to map its local address to PCI Address of RC. (To allow outbound memory write to RC RAM space).

4. On receiving a transmit request from kernel it initiates a DMA memory copy of the packet (in the socket buffer) to the RC memory through the outbound window. After DMA finishes it sends an interrupt to RC through the conventional PCI MSI transaction.

5. On reception of a packet (from RC) the msi interrupt handler is called and it copies the packet in RAM to a socket buffer and passes it to the kernel.

So basically a bidirectional communication channel has been established but the driver is not ready for performance checks yet. I am working on it now. I will report any improvements obtained in this regard.

References: -

1. <https://lists.ozlabs.org/pipermail/linuxppc-dev/2013-September/111069.html>
2. <https://github.com/nxp-auto-linux/auto_yocto_bsp>
3. <https://www.nxp.com/design/design-center/software/embedded-software/linux-software-and-development-tools/bsp-for-s32-microcontrollers-and-processors:BSP-S32>
4. <https://github.com/nxp-auto-linux>