HCI backend for NFC Core

- Author: Eric Lapuyade, Samuel Ortiz
- Contact: eric.lapuyade@intel.com, samuel.ortiz@intel.com

General

The HCI layer implements much of the ETSI TS 102 622 V10.2.0 specification. It enables easy writing of HCI-based NFC drivers. The HCI layer runs as an NFC Core backend, implementing an abstract nfc device and translating NFC Core API to HCI commands and events.

HCI

HCI registers as an nfc device with NFC Core. Requests coming from userspace are routed through netlink sockets to NFC Core and then to HCI. From this point, they are translated in a sequence of HCI commands sent to the HCI layer in the host controller (the chip). Commands can be executed synchronously (the sending context blocks waiting for response) or asynchronously (the response is returned from HCI Rx context). HCI events can also be received from the host controller. They will be handled and a translation will be forwarded to NFC Core as needed. There are hooks to let the HCI driver handle proprietary events or override standard behavior. HCI uses 2 execution contexts:

- one for executing commands : nfc_hci_msg_tx_work(). Only one command can be executing at any given moment.
- one for dispatching received events and commands : nfc_hci_msg_rx_work().

HCI Session initialization

The Session initialization is an HCI standard which must unfortunately support proprietary gates. This is the reason why the driver will pass a list of proprietary gates that must be part of the session. HCI will ensure all those gates have pipes connected when the hci device is set up. In case the chip supports pre-opened gates and pseudo-static pipes, the driver can pass that information to HCI core.

HCI Gates and Pipes

A gate defines the 'port' where some service can be found. In order to access a service, one must create a pipe to that gate and open it. In this implementation, pipes are totally hidden. The public API only knows gates. This is consistent with the driver need to send commands to proprietary gates without knowing the pipe connected to it.

Driver interface

A driver is generally written in two parts : the physical link management and the HCI management. This makes it easier to maintain a driver for a chip that can be connected using various phy (i2c, spi, ...)

HCI Management

A driver would normally register itself with HCI and provide the following entry points:

```
struct nfc_hci_ops {
      int (*open)(struct nfc_hci_dev *hdev);
      void (*close)(struct nfc_hci_dev *hdev);
      int (*hci_ready) (struct nfc_hci_dev *hdev);
      int (*xmit) (struct nfc_hci_dev *hdev, struct sk_buff *skb);
      int (*start_poll) (struct nfc_hci_dev *hdev,
                         u32 im_protocols, u32 tm_protocols);
      int (*dep_link_up)(struct nfc_hci_dev *hdev, struct nfc_target *target,
                         u8 comm_mode, u8 *gb, size_t gb_len);
      int (*dep_link_down)(struct nfc_hci_dev *hdev);
      int (*target_from_gate) (struct nfc_hci_dev *hdev, u8 gate,
                               struct nfc_target *target);
      int (*complete_target_discovered) (struct nfc_hci_dev *hdev, u8 gate,
                                         struct nfc_target *target);
      int (*im_transceive) (struct nfc_hci_dev *hdev,
                            struct nfc_target *target, struct sk_buff *skb,
                            data_exchange_cb_t cb, void *cb_context);
      int (*tm_send)(struct nfc_hci_dev *hdev, struct sk_buff *skb);
      int (*check_presence)(struct nfc_hci_dev *hdev,
                            struct nfc_target *target);
      int (*event_received)(struct nfc_hci_dev *hdev, u8 gate, u8 event,
                            struct sk_buff *skb);
};
```

- open() and close() shall turn the hardware on and off.
- hci_ready() is an optional entry point that is called right after the hci session has been set up. The driver can use it to do additional initialization that must be performed using HCI commands.
- xmit() shall simply write a frame to the physical link.
- start_poll() is an optional entrypoint that shall set the hardware in polling mode. This must be implemented only if the hardware uses proprietary gates or a mechanism slightly different from the HCI standard.
- dep_link_up() is called after a p2p target has been detected, to finish the p2p connection setup with hardware parameters that need to be passed back to nfc core.
- dep_link_down() is called to bring the p2p link down.
- target_from_gate() is an optional entrypoint to return the nfc protocols corresponding to a proprietary gate.
- complete_target_discovered() is an optional entry point to let the driver perform additional proprietary processing necessary to auto activate the discovered target.
- im_transceive() must be implemented by the driver if proprietary HCI commands are required to send data to the tag. Some tag types will require custom commands, others can be written to using the standard HCI commands. The driver can check the tag type and either do proprietary processing, or return 1 to ask for standard processing. The data exchange command itself must be sent asynchronously.
- tm_send() is called to send data in the case of a p2p connection
- check_presence() is an optional entry point that will be called regularly by the core to check that an activated tag is still in the field. If this is not implemented, the core will not be able to push tag_lost events to the user space
- event_received() is called to handle an event coming from the chip. Driver can handle the event or return 1 to let HCI attempt standard processing.

On the rx path, the driver is responsible to push incoming HCP frames to HCI using nfc_hci_recv_frame(). HCI will take care of re-aggregation and handling This must be done from a context that can sleep.

PHY Management

The physical link (i2c, ...) management is defined by the following structure:

```
struct nfc_phy_ops {
      int (*write)(void *dev_id, struct sk_buff *skb);
      int (*enable)(void *dev_id);
      void (*disable)(void *dev_id);
};
```

enable():
: turn the phy on (power on), make it ready to transfer data

disable():
: turn the phy off

write():
: Send a data frame to the chip. Note that to enable higher layers such as an llc to store the frame for re-emission, this function must not alter the skb. It must also not return a positive result (return 0 for success, negative for failure).

Data coming from the chip shall be sent directly to nfc_hci_recv_frame().

LLC

Communication between the CPU and the chip often requires some link layer protocol. Those are isolated as modules managed by the HCI layer. There are currently two modules : nop (raw transfert) and shdlc. A new llc must implement the following functions:

```
struct nfc_llc_ops {
      void *(*init) (struct nfc_hci_dev *hdev, xmit_to_drv_t xmit_to_drv,
                     rcv_to_hci_t rcv_to_hci, int tx_headroom,
                     int tx_tailroom, int *rx_headroom, int *rx_tailroom,
                     llc_failure_t llc_failure);
      void (*deinit) (struct nfc_llc *llc);
      int (*start) (struct nfc_llc *llc);
      int (*stop) (struct nfc_llc *llc);
      void (*rcv_from_drv) (struct nfc_llc *llc, struct sk_buff *skb);
      int (*xmit_from_hci) (struct nfc_llc *llc, struct sk_buff *skb);
};
```

init():
: allocate and init your private storage

deinit():
: cleanup

start():
: establish the logical connection

stop ():
: terminate the logical connection

rcv_from_drv():

handle data coming from the chip, going to HCI

xmit_from_hci():

handle data sent by HCI, going to the chip

The llc must be registered with nfc before it can be used. Do that by calling:

```
nfc_llc_register(const char *name, const struct nfc_llc_ops *ops);
```

Again, note that the llc does not handle the physical link. It is thus very easy to mix any physical link with any llc for a given chip driver.

Included Drivers

An HCI based driver for an NXP PN544, connected through I2C bus, and using shdlc is included.

Execution Contexts

The execution contexts are the following: - IRQ handler (IRQH): fast, cannot sleep. sends incoming frames to HCI where they are passed to the current llc. In case of shdlc, the frame is queued in shdlc rx queue.

- SHDLC State Machine worker (SMW)

  Only when llc_shdlc is used: handles shdlc rx & tx queues.

  Dispatches HCI cmd responses.

- HCI Tx Cmd worker (MSGTXWQ)

  Serializes execution of HCI commands.

  Completes execution in case of response timeout.

- HCI Rx worker (MSGRXWQ)

  Dispatches incoming HCI commands or events.

- Syscall context from a userspace call (SYSCALL)

  Any entrypoint in HCI called from NFC Core

Workflow executing an HCI command (using shdlc)

Executing an HCI command can easily be performed synchronously using the following API:

```
int nfc_hci_send_cmd (struct nfc_hci_dev *hdev, u8 gate, u8 cmd,
                      const u8 *param, size_t param_len, struct sk_buff **skb)
```

The API must be invoked from a context that can sleep. Most of the time, this will be the syscall context. skb will return the result that was received in the response.

Internally, execution is asynchronous. So all this API does is to enqueue the HCI command, setup a local wait queue on stack, and wait_event() for completion. The wait is not interruptible because it is guaranteed that the command will complete after some short timeout anyway.

MSGTXWQ context will then be scheduled and invoke nfc_hci_msg_tx_work(). This function will dequeue the next pending command and send its HCP fragments to the lower layer which happens to be shdlc. It will then start a timer to be able to complete the command with a timeout error if no response arrive.

SMW context gets scheduled and invokes nfc_shdlc_sm_work(). This function handles shdlc framing in and out. It uses the driver xmit to send frames and receives incoming frames in an skb queue filled from the driver IRQ handler. SHDLC I(nformation) frames payload are HCP fragments. They are aggregated to form complete HCI frames, which can be a response, command, or event.

HCI Responses are dispatched immediately from this context to unblock waiting command execution. Response processing involves invoking the completion callback that was provided by nfc_hci_msg_tx_work() when it sent the command. The completion callback will then wake the syscall context.

It is also possible to execute the command asynchronously using this API:

```
static int nfc_hci_execute_cmd_async(struct nfc_hci_dev *hdev, u8 pipe, u8 cmd,
                                     const u8 *param, size_t param_len,
                                     data_exchange_cb_t cb, void *cb_context)
```

The workflow is the same, except that the API call returns immediately, and the callback will be called with the result from the SMW context.

Workflow receiving an HCI event or command

HCI commands or events are not dispatched from SMW context. Instead, they are queued to HCI rx_queue and will be dispatched from HCI rx worker context (MSGRXWQ). This is done this way to allow a cmd or event handler to also execute other commands (for example, handling the NFC_HCI_EVT_TARGET_DISCOVERED event from PN544 requires to issue an ANY_GET_PARAMETER to the reader A gate to get information on the target that was discovered).

Typically, such an event will be propagated to NFC Core from MSGRXWQ context.

Error management

Errors that occur synchronously with the execution of an NFC Core request are simply returned as the execution result of the request. These are easy.

Errors that occur asynchronously (e.g. in a background protocol handling thread) must be reported such that upper layers don't stay ignorant that something went wrong below and know that expected events will probably never happen. Handling of these errors is done as follows:

- driver (pn544) fails to deliver an incoming frame: it stores the error such that any subsequent call to the driver will result in this error. Then it calls the standard nfc_shdlc_recv_frame() with a NULL argument to report the problem above. shdlc stores a EREMOTEIO sticky status, which will trigger SMW to report above in turn.
- SMW is basically a background thread to handle incoming and outgoing shdlc frames. This thread will also check the shdlc sticky status and report to HCI when it discovers it is not able to run anymore because of an unrecoverable error that happened within shdlc or below. If the problem occurs during shdlc connection, the error is reported through the connect completion.
- HCI: if an internal HCI error happens (frame is lost), or HCI is reported an error from a lower layer, HCI will either complete the currently executing command with that error, or notify NFC Core directly if no command is executing.
- NFC Core: when NFC Core is notified of an error from below and polling is active, it will send a tag discovered event with an empty tag list to the user space to let it know that the poll operation will never be able to detect a tag. If polling is not active and the error was sticky, lower levels will return it at next invocation.