CEC Kernel Support

The CEC framework provides a unified kernel interface for use with HDMI CEC hardware. It is designed to handle a multiple types of hardware (receivers, transmitters, USB dongles). The framework also gives the option to decide what to do in the kernel driver and what should be handled by userspace applications. In addition it integrates the remote control passthrough feature into the kernel's remote control framework.

The CEC Protocol

The CEC protocol enables consumer electronic devices to communicate with each other through the HDMI connection. The protocol uses logical addresses in the communication. The logical address is strictly connected with the functionality provided by the device. The TV acting as the communication hub is always assigned address 0. The physical address is determined by the physical connection between devices.

The CEC framework described here is up to date with the CEC 2.0 specification. It is documented in the HDMI 1.4 specification with the new 2.0 bits documented in the HDMI 2.0 specification. But for most of the features the freely available HDMI 1.3a specification is sufficient:

https://www.hdmi.org/spec/index

CEC Adapter Interface

The struct cec_adapter represents the CEC adapter hardware. It is created by calling cec_allocate_adapter() and deleted by calling cec_delete_adapter():

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]cec-core.rst, line 38)

Unknown directive type "c:function".

```
.. c:function::
   struct cec_adapter *cec_allocate_adapter(const struct cec_adap_ops *ops, \
                                            void *priv, const char *name, \
                                            u32 caps, u8 available_las);
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]cec-core.rst, line 43)

Unknown directive type "c:function".

```
.. c:function::
   void cec_delete_adapter(struct cec_adapter *adap);
```

To create an adapter you need to pass the following information:

ops:
: adapter operations which are called by the CEC framework and that you have to implement.

priv:
: will be stored in adap->priv and can be used by the adapter ops. Use cec_get_drvdata(adap) to get the priv pointer.

name:
: the name of the CEC adapter. Note: this name will be copied.

caps:
: capabilities of the CEC adapter. These capabilities determine the capabilities of the hardware and which parts are to be handled by userspace and which parts are handled by kernelspace. The capabilities are returned by CEC_ADAP_G_CAPS.

available_las:
: the number of simultaneous logical addresses that this adapter can handle. Must be 1 <= available_las <= CEC_MAX_LOG_ADDRS.

To obtain the priv pointer use this helper function:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]cec-core.rst, line 71)

Unknown directive type "c:function".

```
.. c:function::
        void *cec_get_drvdata(const struct cec_adapter *adap);
```

To register the /dev/cecX device node and the remote control device (if CEC_CAP_RC is set) you call:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]cec-core.rst, line 77)

Unknown directive type "c:function".

```
.. c:function::
        int cec_register_adapter(struct cec_adapter *adap, \
                                 struct device *parent);
```

where parent is the parent device.

To unregister the devices call:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]cec-core.rst, line 85)

Unknown directive type "c:function".

```
.. c:function::
        void cec_unregister_adapter(struct cec_adapter *adap);
```

Note: if cec_register_adapter() fails, then call cec_delete_adapter() to clean up. But if cec_register_adapter() succeeded, then only call cec_unregister_adapter() to clean up, never cec_delete_adapter(). The unregister function will delete the adapter automatically once the last user of that /dev/cecX device has closed its file handle.

Implementing the Low-Level CEC Adapter

The following low-level adapter operations have to be implemented in your driver:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]cec-core.rst, line 101)

Unknown directive type "c:struct".

```
.. c:struct:: cec_adap_ops
```

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]cec-core.rst, line 103)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

        struct cec_adap_ops
        {
                /* Low-level callbacks */
                int (*adap_enable)(struct cec_adapter *adap, bool enable);
                int (*adap_monitor_all_enable)(struct cec_adapter *adap, bool enable);
                int (*adap_monitor_pin_enable)(struct cec_adapter *adap, bool enable);
                int (*adap_log_addr)(struct cec_adapter *adap, u8 logical_addr);
                int (*adap_transmit)(struct cec_adapter *adap, u8 attempts,
                                     u32 signal_free_time, struct cec_msg *msg);
                void (*adap_status)(struct cec_adapter *adap, struct seq_file *file);
                void (*adap_free)(struct cec_adapter *adap);

                /* Error injection callbacks */
                ...

                /* High-level callbacks */
```

```
                ...
        };
```

The seven low-level ops deal with various aspects of controlling the CEC adapter hardware:

To enable/disable the hardware:

```
int (*adap_enable)(struct cec_adapter *adap, bool enable);
```

This callback enables or disables the CEC hardware. Enabling the CEC hardware means powering it up in a state where no logical addresses are claimed. The physical address will always be valid if CEC_CAP_NEEDS_HPD is set. If that capability is not set, then the physical address can change while the CEC hardware is enabled. CEC drivers should not set CEC_CAP_NEEDS_HPD unless the hardware design requires that as this will make it impossible to wake up displays that pull the HPD low when in standby mode. The initial state of the CEC adapter after calling cec_allocate_adapter() is disabled.

Note that adap_enable must return 0 if enable is false.

To enable/disable the 'monitor all' mode:

```
int (*adap_monitor_all_enable)(struct cec_adapter *adap, bool enable);
```

If enabled, then the adapter should be put in a mode to also monitor messages that are not for us. Not all hardware supports this and this function is only called if the CEC_CAP_MONITOR_ALL capability is set. This callback is optional (some hardware may always be in 'monitor all' mode).

Note that adap_monitor_all_enable must return 0 if enable is false.

To enable/disable the 'monitor pin' mode:

```
int (*adap_monitor_pin_enable)(struct cec_adapter *adap, bool enable);
```

If enabled, then the adapter should be put in a mode to also monitor CEC pin changes. Not all hardware supports this and this function is only called if the CEC_CAP_MONITOR_PIN capability is set. This callback is optional (some hardware may always be in 'monitor pin' mode).

Note that adap_monitor_pin_enable must return 0 if enable is false.

To program a new logical address:

```
int (*adap_log_addr)(struct cec_adapter *adap, u8 logical_addr);
```

If logical_addr == CEC_LOG_ADDR_INVALID then all programmed logical addresses are to be erased. Otherwise the given logical address should be programmed. If the maximum number of available logical addresses is exceeded, then it should return -ENXIO. Once a logical address is programmed the CEC hardware can receive directed messages to that address.

Note that adap_log_addr must return 0 if logical_addr is CEC_LOG_ADDR_INVALID.

To transmit a new message:

```
int (*adap_transmit)(struct cec_adapter *adap, u8 attempts,
                     u32 signal_free_time, struct cec_msg *msg);
```

This transmits a new message. The attempts argument is the suggested number of attempts for the transmit.

The signal_free_time is the number of data bit periods that the adapter should wait when the line is free before attempting to send a message. This value depends on whether this transmit is a retry, a message from a new initiator or a new message for the same initiator. Most hardware will handle this automatically, but in some cases this information is needed.

The CEC_FREE_TIME_TO_USEC macro can be used to convert signal_free_time to microseconds (one data bit period is 2.4 ms).

To log the current CEC hardware status:

```
void (*adap_status)(struct cec_adapter *adap, struct seq_file *file);
```

This optional callback can be used to show the status of the CEC hardware. The status is available through debugfs: cat /sys/kernel/debug/cec/cecX/status

To free any resources when the adapter is deleted:

```
void (*adap_free)(struct cec_adapter *adap);
```

This optional callback can be used to free any resources that might have been allocated by the driver. It's called from cec_delete_adapter.

Your adapter driver will also have to react to events (typically interrupt driven) by calling into the framework in the following situations:

When a transmit finished (successfully or otherwise):

```
void cec_transmit_done(struct cec_adapter *adap, u8 status,
                       u8 arb_lost_cnt,  u8 nack_cnt, u8 low_drive_cnt,
                       u8 error_cnt);
```

or:

```
void cec_transmit_attempt_done(struct cec_adapter *adap, u8 status);
```

The status can be one of:

CEC_TX_STATUS_OK:
: the transmit was successful.

CEC_TX_STATUS_ARB_LOST:
: arbitration was lost: another CEC initiator took control of the CEC line and you lost the arbitration.

CEC_TX_STATUS_NACK:
: the message was nacked (for a directed message) or acked (for a broadcast message). A retransmission is needed.

CEC_TX_STATUS_LOW_DRIVE:
: low drive was detected on the CEC bus. This indicates that a follower detected an error on the bus and requested a retransmission.

CEC_TX_STATUS_ERROR:
: some unspecified error occurred: this can be one of ARB_LOST or LOW_DRIVE if the hardware cannot differentiate or something else entirely. Some hardware only supports OK and FAIL as the result of a transmit, i.e. there is no way to differentiate between the different possible errors. In that case map FAIL to CEC_TX_STATUS_NACK and not to CEC_TX_STATUS_ERROR.

CEC_TX_STATUS_MAX_RETRIES:
: could not transmit the message after trying multiple times. Should only be set by the driver if it has hardware support for retrying messages. If set, then the framework assumes that it doesn't have to make another attempt to transmit the message since the hardware did that already.

The hardware must be able to differentiate between OK, NACK and 'something else'.

The *_cnt arguments are the number of error conditions that were seen. This may be 0 if no information is available. Drivers that do not support hardware retry can just set the counter corresponding to the transmit error to 1, if the hardware does support retry then either set these counters to 0 if the hardware provides no feedback of which errors occurred and how many times, or fill in the correct values as reported by the hardware.

Be aware that calling these functions can immediately start a new transmit if there is one pending in the queue. So make sure that the hardware is in a state where new transmits can be started *before* calling these functions.

The cec_transmit_attempt_done() function is a helper for cases where the hardware never retries, so the transmit is always for just a single attempt. It will call cec_transmit_done() in turn, filling in 1 for the count argument corresponding to the status. Or all 0 if the status was OK.

When a CEC message was received:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]cec-core.rst, line 281)

Unknown directive type "c:function".

```
.. c:function::
        void cec_received_msg(struct cec_adapter *adap, struct cec_msg *msg);
```

Speaks for itself.

Implementing the interrupt handler

Typically the CEC hardware provides interrupts that signal when a transmit finished and whether it was successful or not, and it provides and interrupt when a CEC message was received.

The CEC driver should always process the transmit interrupts first before handling the receive interrupt. The framework expects to see the cec_transmit_done call before the cec_received_msg call, otherwise it can get confused if the received message was in reply to the transmitted message.

Optional: Implementing Error Injection Support

If the CEC adapter supports Error Injection functionality, then that can be exposed through the Error Injection callbacks:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api]

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]cec-core.rst, line 304)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

        struct cec_adap_ops {
                /* Low-level callbacks */
                ...

                /* Error injection callbacks */
                int (*error_inj_show)(struct cec_adapter *adap, struct seq_file *sf);
                bool (*error_inj_parse_line)(struct cec_adapter *adap, char *line);

                /* High-level CEC message callback */
                ...
        };
```

If both callbacks are set, then an `error-inj` file will appear in debugfs. The basic syntax is as follows:

Leading spaces/tabs are ignored. If the next character is a `#` or the end of the line was reached, then the whole line is ignored. Otherwise a command is expected.

This basic parsing is done in the CEC Framework. It is up to the driver to decide what commands to implement. The only requirement is that the command `clear` without any arguments must be implemented and that it will remove all current error injection commands.

This ensures that you can always do `echo clear >error-inj` to clear any error injections without having to know the details of the driver-specific commands.

Note that the output of `error-inj` shall be valid as input to `error-inj`. So this must work:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]cec-core.rst, line 335)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

        $ cat error-inj >einj.txt
        $ cat einj.txt >error-inj
```

The first callback is called when this file is read and it should show the current error injection state:

```
int (*error_inj_show)(struct cec_adapter *adap, struct seq_file *sf);
```

It is recommended that it starts with a comment block with basic usage information. It returns 0 for success and an error otherwise.

The second callback will parse commands written to the `error-inj` file:

```
bool (*error_inj_parse_line)(struct cec_adapter *adap, char *line);
```

The `line` argument points to the start of the command. Any leading spaces or tabs have already been skipped. It is a single line only (so there are no embedded newlines) and it is 0-terminated. The callback is free to modify the contents of the buffer. It is only called for lines containing a command, so this callback is never called for empty lines or comment lines.

Return true if the command was valid or false if there were syntax errors.

Implementing the High-Level CEC Adapter

The low-level operations drive the hardware, the high-level operations are CEC protocol driven. The following high-level callbacks are available:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]cec-core.rst, line 366)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

        struct cec_adap_ops {
                /* Low-level callbacks */
                ...
```

```
                /* Error injection callbacks */
                ...

                /* High-level CEC message callback */
                int (*received)(struct cec_adapter *adap, struct cec_msg *msg);
        };
```

The received() callback allows the driver to optionally handle a newly received CEC message:

```
int (*received)(struct cec_adapter *adap, struct cec_msg *msg);
```

If the driver wants to process a CEC message, then it can implement this callback. If it doesn't want to handle this message, then it should return -ENOMSG, otherwise the CEC framework assumes it processed this message and it will not do anything with it.

CEC framework functions

CEC Adapter drivers can call the following CEC framework functions:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]cec-core.rst, line 395)

Unknown directive type "c:function".

```
.. c:function::
   int cec_transmit_msg(struct cec_adapter *adap, struct cec_msg *msg, \
                        bool block);
```

Transmit a CEC message. If block is true, then wait until the message has been transmitted, otherwise just queue it and return.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]cec-core.rst, line 402)

Unknown directive type "c:function".

```
.. c:function::
   void cec_s_phys_addr(struct cec_adapter *adap, u16 phys_addr, bool block);
```

Change the physical address. This function will set adap->phys_addr and send an event if it has changed. If cec_s_log_addrs() has been called and the physical address has become valid, then the CEC framework will start claiming the logical addresses. If block is true, then this function won't return until this process has finished.

When the physical address is set to a valid value the CEC adapter will be enabled (see the adap_enable op). When it is set to CEC_PHYS_ADDR_INVALID, then the CEC adapter will be disabled. If you change a valid physical address to another valid physical address, then this function will first set the address to CEC_PHYS_ADDR_INVALID before enabling the new physical address.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]cec-core.rst, line 417)

Unknown directive type "c:function".

```
.. c:function::
   void cec_s_phys_addr_from_edid(struct cec_adapter *adap, \
                                  const struct edid *edid);
```

A helper function that extracts the physical address from the edid struct and calls cec_s_phys_addr() with that address, or CEC_PHYS_ADDR_INVALID if the EDID did not contain a physical address or edid was a NULL pointer.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]cec-core.rst, line 425)

Unknown directive type "c:function".

```
.. c:function::
```

```
int cec_s_log_addrs(struct cec_adapter *adap, \
                    struct cec_log_addrs *log_addrs, bool block);
```

Claim the CEC logical addresses. Should never be called if CEC_CAP_LOG_ADDRS is set. If block is true, then wait until the logical addresses have been claimed, otherwise just queue it and return. To unconfigure all logical addresses call this function with log_addrs set to NULL or with log_addrs->num_log_addrs set to 0. The block argument is ignored when unconfiguring. This function will just return if the physical address is invalid. Once the physical address becomes valid, then the framework will attempt to claim these logical addresses.

CEC Pin framework

Most CEC hardware operates on full CEC messages where the software provides the message and the hardware handles the low-level CEC protocol. But some hardware only drives the CEC pin and software has to handle the low-level CEC protocol. The CEC pin framework was created to handle such devices.

Note that due to the close-to-realtime requirements it can never be guaranteed to work 100%. This framework uses highres timers internally, but if a timer goes off too late by more than 300 microseconds wrong results can occur. In reality it appears to be fairly reliable.

One advantage of this low-level implementation is that it can be used as a cheap CEC analyser, especially if interrupts can be used to detect CEC pin transitions from low to high or vice versa.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]cec-core.rst, line 455)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/media/cec-pin.h
```

CEC Notifier framework

Most drm HDMI implementations have an integrated CEC implementation and no notifier support is needed. But some have independent CEC implementations that have their own driver. This could be an IP block for an SoC or a completely separate chip that deals with the CEC pin. For those cases a drm driver can install a notifier and use the notifier to inform the CEC driver about changes in the physical address.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]cec-core.rst, line 467)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/media/cec-notifier.h
```