Nuxitron Remote Control Interface

Legends:

MCU = Microcontroller unit
config_data = mapping table
EEPROM = Holds the mapping table
panel = In the context of this document panel means TV
i2c = Utility program that uses remote control driver

GOALS

*) New panels are added to the system by changing only file(s).
*) In order to realize remote intervention, file(s) should be transferred to the target place over the network and then written appropriately. This way, support team may handle the situation remotely...

DESCRIPTION

Provides remote control key events to the application and panel. System generally consists of two parts:

- Panel
- PC

Both sides works independently and relies on config_data. The modus operandi of this system is that if you give config_data, you get remote control key events...

Panel side of the system starts to work when the power is applied and config_data is valid. Thus, someone can control the panel immediately...

PC side starts to work when the driver is opened for reading and config_data is valid. Thus, buffering problems are avoided when the PC side is inactive...

Because both sides rely on config_data, the first thing to be done is to write config_data correctly, then remote control key events is ready to be delivered to both sides. After that whenever power is applied, as the first job, EEPROM is read...

[open and close routines]

Description

In order to open the device, "/dev/remote" node should have been created.
remote MAJOR -> 241
remote MINOR -> 0
Or major and minor numbers can be specified at module load time. Module parameter names:
- major
- minor

Return Value

0 is returned...

Notes

* If two or more reader(consumer) thread try to open it, may block...
* reader threads(processes) are tracked down automatically. Let's say there are two open copies of the file. One is for reading and opened in the read-only mode, the other one is for writing and opened in the read-write mode. Then this means two reader threads are exist. If the read-only copy is closed somehow, PC side continues being active. So afterwards the driver is opened for getting remote control key events, may contain stale data...

I can replace the above thema with something different.

[write routine] -> i2c :: static int config_data_write_cmd(int device_fd)

Utility pair

./i2c configwrite irda panel

Description

Write routine triggers config_data_write sequence. config_data is represented with the following structure:

```
/* Dont leave out the alignment to the compiler */
struct eeprom_table {
        __u8 reserved[8];
        __u32 panel_irda_key[PANEL_ENTRY_SIZE];
        __u16 panel_wait_time[PANEL_ENTRY_SIZE];
        __u8 procedure[ONE_TO_MUL_MAPPING_SIZE];
        __u8 irda_coding[IRDA_CODING_SIZE];
        __u8 padding[2];
        __u32 irda_tab[IRDA_KEY_SIZE]; /* iNUX irda_key table */
} __attribute__ ((packed)) e2prom_tab;
```

And this structure is fed with two files in the example utility(i2c): irda and panel.

irda - Holds iNUX remote control irda keys
panel - Holds panel irda keys

As long as you fill the above structure correctly, any kind of file format can be chosen. Only restriction is that the order in the irda and panel files must be followed...

Because EEPROM holds its state when the power is off, config_data is only written once in the installation. After that, it doesnt need to be written. But if there is any problem in the system and it is solved only by updating e2prom, config_data has to be written once again...

Return Value

On success; "sizeof(e2prom_tab)" is returned...

On failure;

* when -EINVAL is returned, this means that provided e2prom table size is not equal to the driver's view.
* when -EFAULT is returned, this means that e2prom_tab is not copied to kernel or user space completely / correctly.
* when -EIO is returned, this means that EEPROM is not accessed.

Notes

* If the return value is different from "sizeof(e2prom_tab)", this means that some kind of error has occured and irda keys may not(-EIO) be get or correct until the config_data is stored to e2prom successfully...
* After config_data is written successfully, system becomes active. There is no additional step...
* May block while writing to the EEPROM...

[read routine] -> i2c :: static int read_device_cmd(int device_fd) :: static int read_file_cmd(int device_fd)

Utility pair

./i2c readdev
./i2c readfile irda

Description

Read routine supplies remote control key events. If there is no data, blocks the calling process...

Example Usage

```
__u32 irda_data = 0;

result = read(device_fd, &irda_data, sizeof(irda_data));
if (result < 0)
        fprintf(stderr, "i2c: cannot read the device\n");
else if (result == 0)
        fprintf(stderr, "i2c: there is no data\n");
else {
        printf("i2c: irda_data = %X\n", irda_data);
        lookup(irda_data);
}
```

Return Value

On success; irda_key is returned, which is 4 bytes in size. File called “irda” holds the iNUX remote control irda keys.

On failure;

* when -EAGAIN is returned, this means that the driver is opened with O_NONBLOCK is set.

* when -EINVAL is returned, this means that at once more than 4 byte is tried to read.

* when -EFAULT is returned, this means that e2prom_tab is not copied to kernel or user space completely / correctly.

* when 0 is returned, this means that config_data cannot be read after power-up.

Notes

./i2c readdev method may be prefferred...

[ioctl routines]

Utility pair -> i2c :: static int config_data_read_cmd(int device_fd)

./i2c configread

Description

Provides the in-memory config_data...

Example Usage

ioctl(device_fd, REMOTE_READ_CONFIG_DATA_CMD, &e2prom_tab);

Return Value

On success; "sizeof(e2prom_tab)" is returned...

On failure;

* when -EPERM is returned, this means that program has not got the required rights. Only root can do that.

* when -EFAULT is returned, this means that e2prom_tab is not copied to kernel or user space completely / correctly.

++

Utility pair -> i2c :: static int welcome_cmd(int device_fd)

./i2c welcome

Description

Opens the panel when the customer arrives the room first time. Some hotels want this feature. When the customer comes to hotel first time and after did the check-in in the reception, they want the panel be open...

Example Usage

ioctl(device_fd, REMOTE_READ_CONFIG_DATA_CMD, &e2prom_tab);

Return Value

On success; a positive value(1) is returned...

On failure;

* when -EPERM is returned, this means that program has not got the required rights. Only root can do that.

* when -EIO is returned, this means that MCU is not accessed.

++

Utility pair -> i2c :: static int start_cmd(int device_fd) :: static int stop_cmd(int device_fd)

./i2c start

./i2c stop

Description

./i2c start -> Tell the MCU start to send data(iNUX irda_key)...

./i2c stop -> Tell the MCU stop to send data(iNUX irda_key)...

Example Usage

ioctl(device_fd, REMOTE_START_CMD, 0);

ioctl(device_fd, REMOTE_STOP_CMD, 0);

Return Value

On success; a positive value(1) is returned...

On failure;

* when -EPERM is returned, this means that program has not got the required rights. Only root can do that.

* when -EIO is returned, this means that MCU is not accessed.

++

Utiliy pair -> i2c :: static int test_mcu_read_cmd(int device_fd) :: static int test_mcu_write_cmd(int device_fd)

./i2c mcuread len

./i2c mcuwrite len val

Description

Reads from and writes to the MCU. Debugging purposes...

Example Usage

ioctl(device_fd, REMOTE_READ_MCU_CMD, &param.io);

ioctl(device_fd, REMOTE_WRITE_MCU_CMD, &param.io);

Return Value

On success; a positive value(1) is returned...

On failure;

* when -EPERM is returned, this means that program has not got the required rights. Only root can do that.

* when -EFAULT is returned, this means that param.io is not copied to kernel or user space completely / correctly.

* when -EINVAL is returned, this means that read or write size is bigger than the max. value.

```
#define MCU_READ_SIZE_MAX  200
#define MCU_WRITE_SIZE_MAX          200
```

* when -EIO is returned, this means that MCU is not accessed.

++

Utiliy pair -> i2c :: static int test_eeprom_read_cmd(int device_fd) :: static int test_eeprom_write_cmd(int device_fd)

./i2c e2read addr len

./i2c e2write addr len val

Description

Reads from and writes to the EEPROM. Debugging purposes...

Example Usage

ioctl(device_fd, REMOTE_READ_EEPROM_CMD, &param.io);

ioctl(device_fd, REMOTE_WRITE_EEPROM_CMD, &param.io);

Return Value

On success; a positive value is returned...

On failure;

* when -EPERM is returned, this means that program has not got the required rights. Only root can do that.

* when -EFAULT is returned, this means that param.io is not copied to kernel or user space completely / correctly.

* when -EINVAL is returned, this means that read or write size is bigger than the max. value.

read - #define EEPROM_SIZE 0x200

write - #define EEPROM_PAGE_WRITE_SIZE 16

* when -EIO is returned, this means that EEPROMs is not accessed.

Notes

“e2write” writes 16 byte at most. If you want to write more than 16 byte, you have to dealt with explicitly. “configwrite” deals with them internally...

When the e2prom is written with e2write, config_data is accepted “stale”. So if you want to continue reading remote control key events, config_data has to be updated...