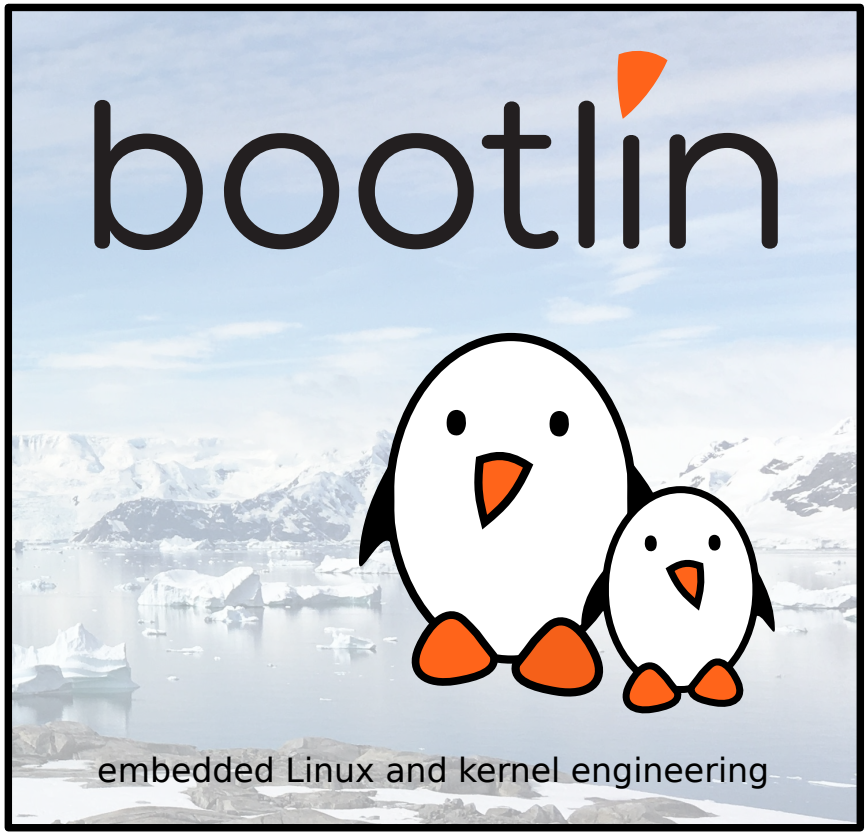
I3C in tomorrow's designs

Miquèl Raynal

miquel@bootlin.com



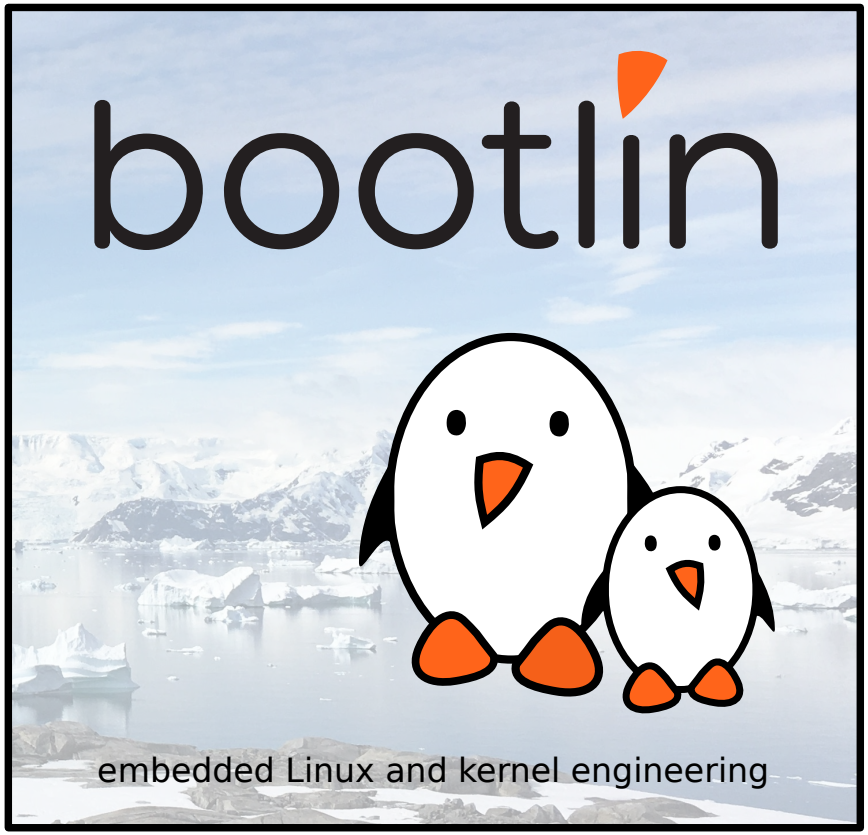
Miquèl Raynal

- Embedded Linux engineer at Bootlin
  - Embedded Linux **expertise**
  - **Development**, consulting and training
  - Strong open-source focus
  - `https://bootlin.com`
- Contributions
  - **Maintainer of the NAND subsystem**
  - **Co-maintainer of the MTD subsystem**
  - **Kernel support for various ARM SoCs**
  - **Contributor to the I3C sybsystem**
- Living in **Toulouse**, south west of France

Yet another serial bus?

Miquèl Raynal

miquel@bootlin.com



Electrical engineering 101

You are a hardware designer.

You want to wire a simple device to a custom board.

What bus do you pick?

Electrical engineering 101

- Existing device interfaces?
- Power consumption targets?
- Expected throughput?
- Wiring constraints?
- Number of devices?

- Pick one:
  - OneWire? Too simple/slow?
  - RS232? Really?
  - USB? Really suitable for embedded systems?
  - I2C? Why not?
  - SPI? Yeah, still trendy?

Electrical engineering 101

You are a hardware designer.

Now you want to wire several devices.
You have minimal bandwidth constraints.
The devices are a bit complex.

What bus do you pick?

State of the art

- I2C is pretty slow (400kHz); SPI is fast (up to 100MHz)
  - Slowest exchanges will last longer
  - Longer exchanges will increase the power consumed per bit transmitted ratio
  - Slow bus with many devices means extra delays
- SPI needs one physical CS **per device**; I2C doesn't, but suffers from address collision issues (lack of virtual addressing)
  - Extra wires on a serial bus are costly
  - I2C won't natively support more than a couple of identical devices
  - In both cases, if asynchronous signaling is needed (IRQ), an additional line must be wired
- Devices not available anymore on the bus after entering low power modes
- Devices don't natively belong to a particular "class" sharing a set of properties (easing the tuning of a set of devices)

Here comes I3C

- **Improved** Inter Integrated Circuit
- MIPI Alliance specification
- Targets automotive, consumer electronics, IoT market
- Two-wire bus with the following main features:
  - Reasonably high throughput
  - Colorful set of commands
  - Dynamic address assignment
  - Self-describing devices (USB-like)
  - In-band signaling
  - Hot-join
  - Controller handover
  - Advanced low-power modes
  - And more...
- Partially backward compatible with I2C

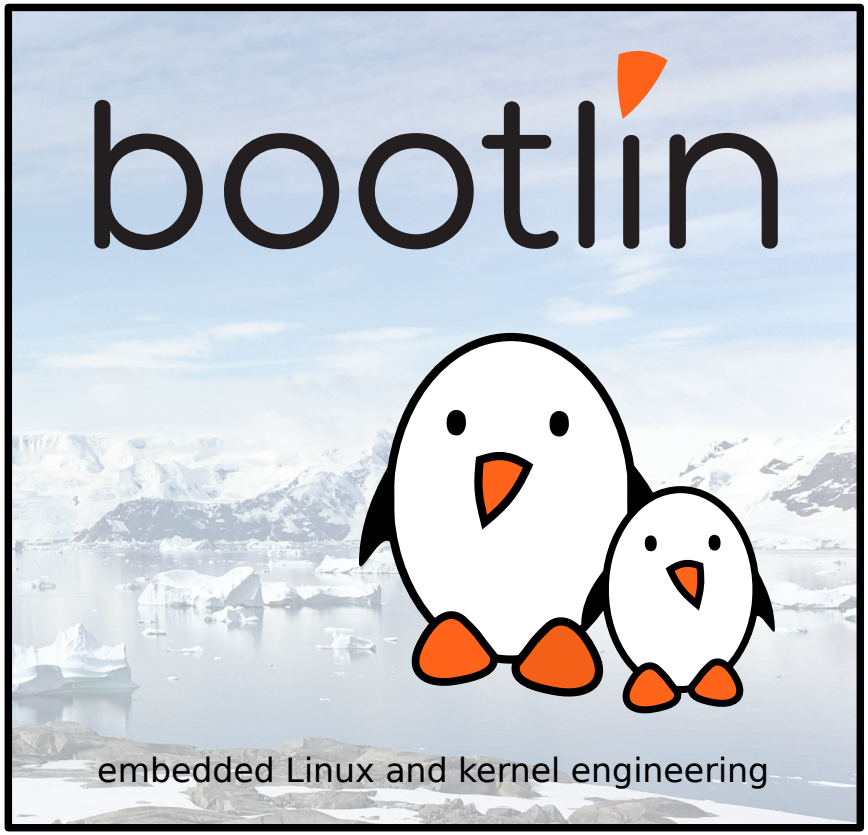
Backward compatibility with I2C

- Main goal:
  - Ease acceptance
  - Improve the re-usability of existing devices
- Designed with I2C in mind
  - At physical level:
    - Compatible Voltage levels
    - A clock and a data line
    - Adaptable clock frequencies
    - Support for open-drain output
  - At logical level:
    - 7-bits addressing
    - Comparable `S`/`Sr`/`P`/`ACK`/`NACK` behavior
  - Use tricks to talk to I2C and I3C devices on the same bus

The I3C protocol

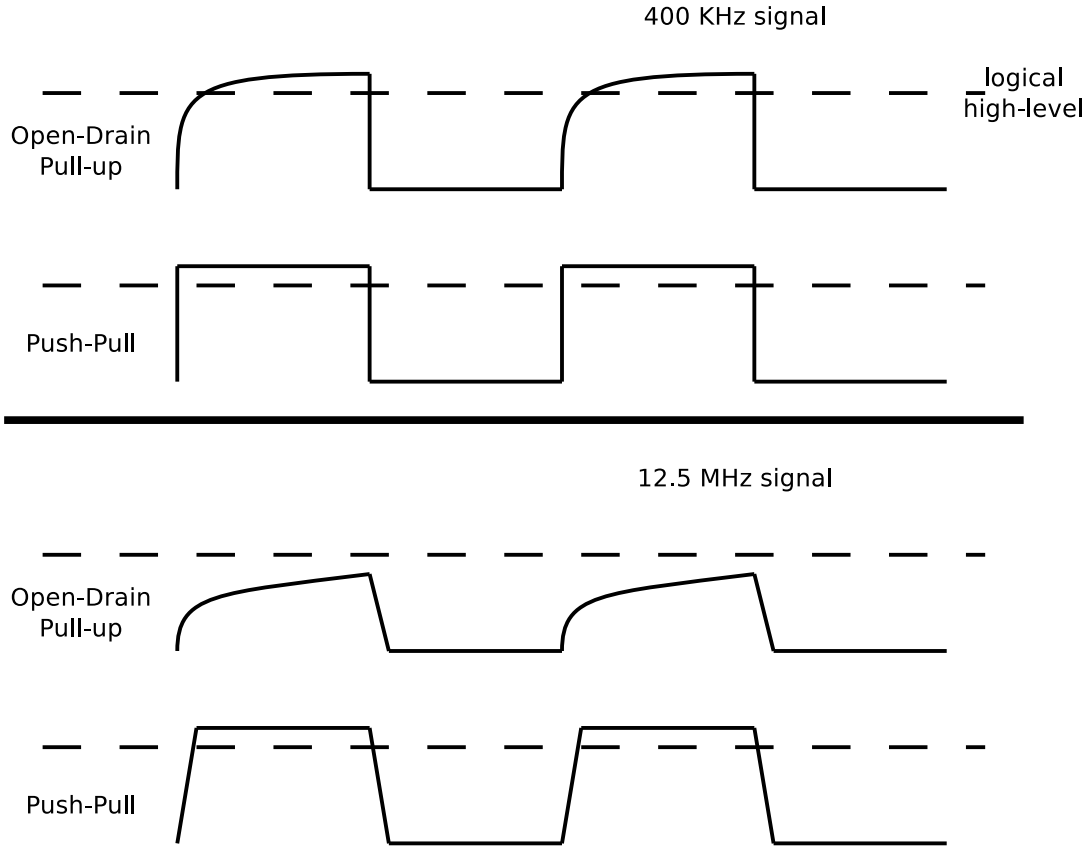
Miquèl Raynal

miquel@bootlin.com



Achieving higher frequencies

- I2C relies on open-drain outputs with pull-ups
- I3C relies on push-pull mode
  - SCL always in push-pull mode
  - SDA alternating push-pull and open-drain



⇒ faster rising times

Pure I3C frequencies

I3C bus typical frequency is 12.5MHz. A "Pure Bus" supports:

- `SDR` (Simple Data Rate) mode achieves 12.5Mpbs
- `SDR` is the default mode and should be supported by all I3C devices
- Optional `HDR` (High Data Rate) modes
  - Double Data Rate (`HDR-DDR`) achieves 25Mbps
    - Still uses the clock
    - Samples data on both edges of the clock
  - Ternary Symbol Pure (`HDR-TSP`) achieves 33.3Mbps
    - Both signals are considered data
    - Tracks edges rather than levels
    - Each transition is a clock cycle, the bus is capable of expressing three states: `SCL` transitioned, `SDA` transitioned, `SCL` and `SDA` transitioned

The MIPI alliance states that higher frequencies only work well with point-to-point topologies:
`https://www.mipi.org/resources/I3C-frequently-asked-questions`

Pure I3C frequencies

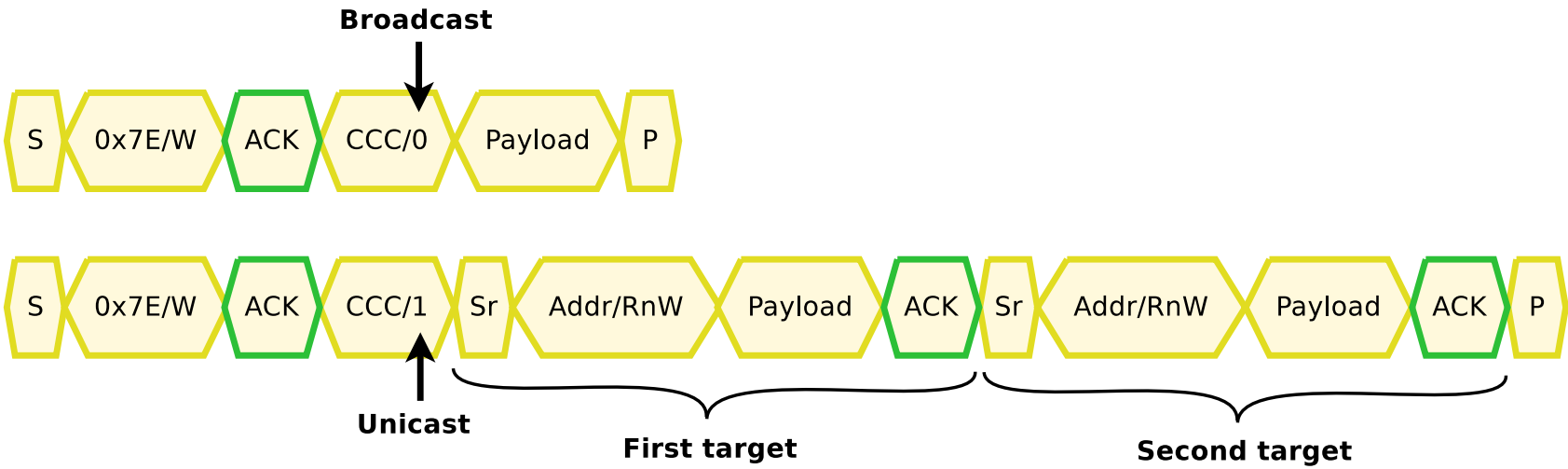
- The controller must advertise all targets that it is entering a HDR mode and which mode it has chosen
- If a device does not support the chosen HDR mode that the controller is entering, it can simply wait for the HDR mode to be exited
- All targets should support tracking the HDR exit pattern
  - 4 SDA fall transitions with SCL low before a P pattern
  - The HDR exit pattern detector is very cheap to implement (simply with 4 latches in series)

Bandwidth: mixed modes

- SCL being configured in push-pull, clock stretching is not allowed
- I2C devices must not be confused by the high-frequency exchanges
  - Most I2C devices already have a 50ns anti-glitch spike filter
  - I3C max bus frequency: 12.5MHz ⇒ 80ns clock period
  - When talking at 12.5MHz I2C devices won't even notice clock changes
  - Asymmetrical clock: less than 50% duty cycle in order to keep the low state longer than 50ns and be sure to be filtered out before reaching the I2C device
  - A "Mixed Fast Bus" supports the following clock modes:
    - SDR
    - HDR-DDR
    - Ternary Symbol Legacy (HDR-TSL) in place of HDR-TSP
- If an I2C device has no spike filter, it's considered a "Mixed Slow/Limited bus", the bus may only work at the slowest device speed

Enhanced bus management

- Common Command Codes (CCC)
- Standardization of the bus management
- Serve all kind of purpose
- Supports broadcast and direct addressing
- CCCs are always sent to the broadcast address
  - 0x7E is the I3C broadcast address
  - 0x7E is reserved in the I2C specification
  - All I3C devices should listen at the entire command



Common Command Codes (CCC)

- ENTDAA: Start a DAA procedure (auto-discovery procedure)
- ENTASX: Enter Activity State (related to power management)
- GETPID: Get Provisional ID (related to device identification)
- GETBCR: Get Bus Characteristics Register (related to device capabilities)
- GETDCR: Get Device Characteristics Register (related to device classification)

ENEC, DISEC, ENTAS0, ENTAS1, ENTAS2, ENTAS3, RSTDAA, ENTDAA, DEFSLVS, SETMWL, SETMRL, ENTTM, ENTHDR0, ENTHDR1, ENTHDR2, ENTHDR3, ENTHDR7, SETXTIME, ENEC, DISEC, ENTAS0, ENTAS1, ENTAS2, ENTAS3, RSTDAA, SETDASA, SETNEWDA, SETMWL, SETMRL, GETMWL, GETMRL, GETPID, GETBCR, GETDCR, GETSTATUS, GETACCMST, SETBRGTGT, GETMXDS, GETHDRCAP, SETXTIME, GETXTIME...

Dynamic Address Assignment (DAA)

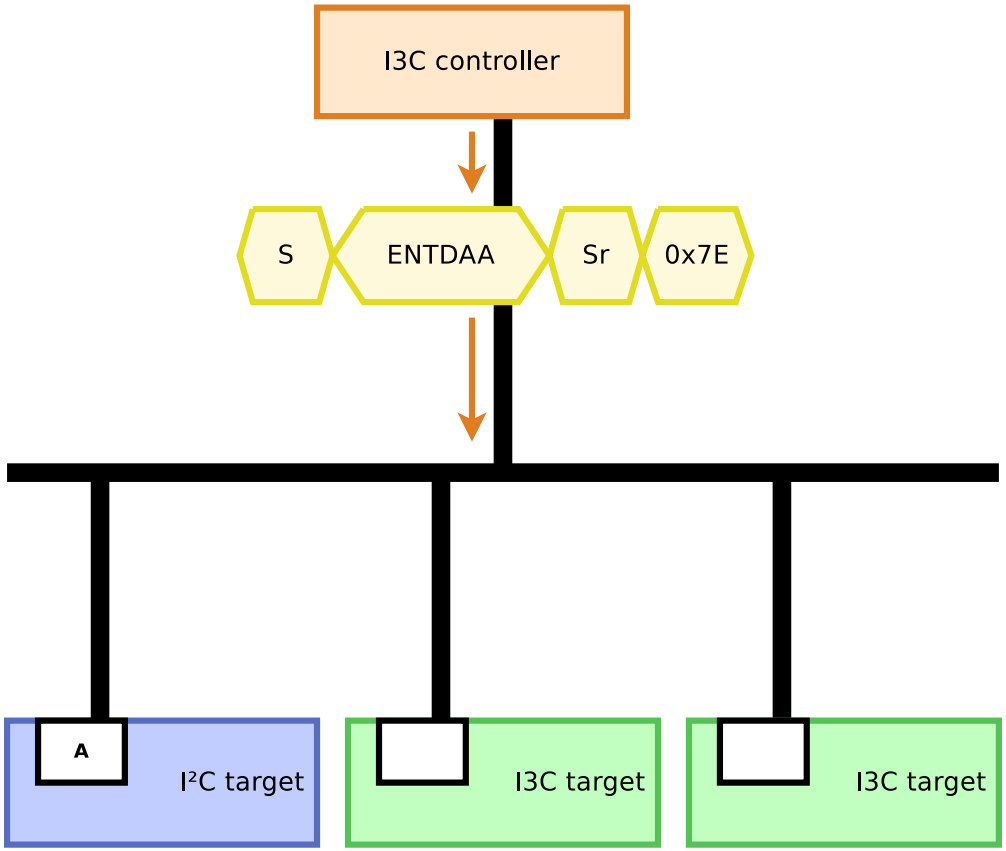
- Regular exchanges on the I3C bus require a dynamic 7-bit address
- I2C devices already have one and the controller knows about them
  - Must be statically described, like any regular I2C device
- I3C devices must get part of the DAA procedure in order to receive a 7-bit dynamic identifier
  - Unless they have a static address as well and the controller knows about it
  - Or they are in a deep sleep state/powered down at that moment
    - They will require the DAA procedure to be resumed when they will become available

Device unique identification

- Each device has its own unique Provisional ID (PID) 48-bit identifier:
  - 15 bits for the MIPI manufacturer ID
  - 1 bit indicating if the below values are random or not
  - 32 bits for the Part ID, Instance ID and more bits left to the vendor to define
- Device also advertise a Bus Characteristics Register BCR which contains:
  - The device role (controller/target)
  - Its potential SDR limitations
  - Its HDR capabilities
  - Its offline capabilities
  - Its in-band signaling capabilities
- It also has a Device Characteristics Register DCR which describes the type of device (accelerometer, thermometer, composite, etc)
  - This calls for standardized interfaces!
- PID, BCR and DCR will be advertised during the DAA procedure

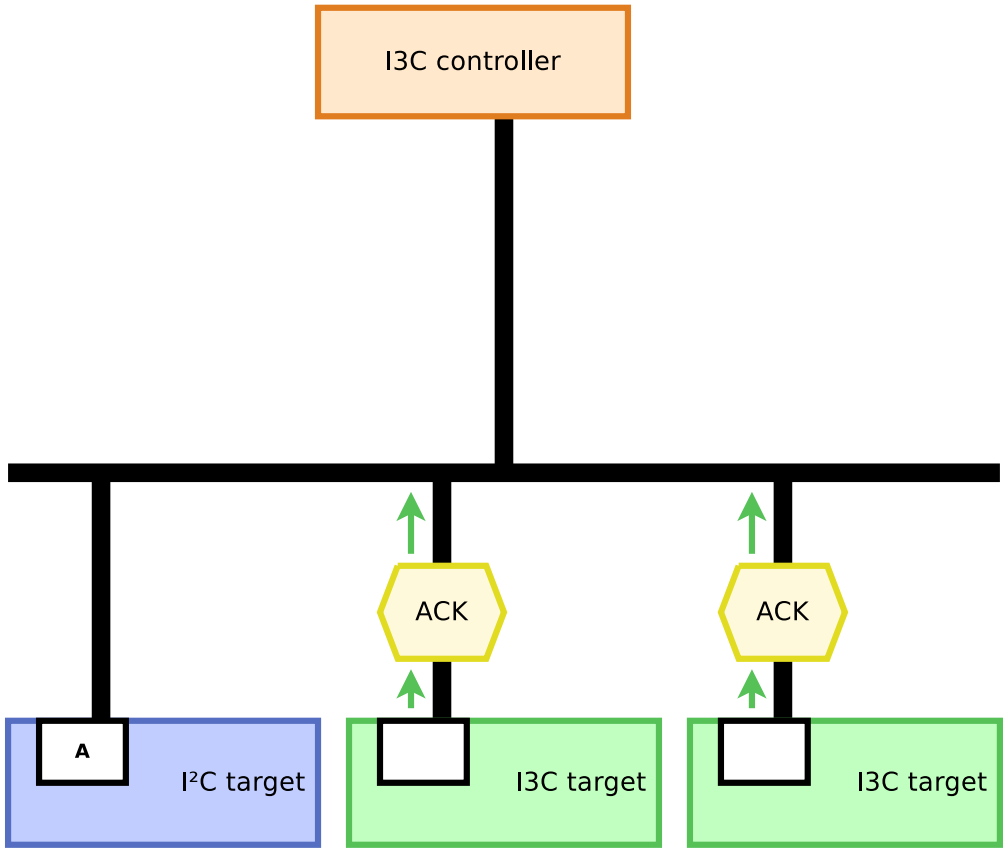
DAA

The controller sends `ENTDAA`, generates a `Sr` and sends the broadcast address (0x7E).

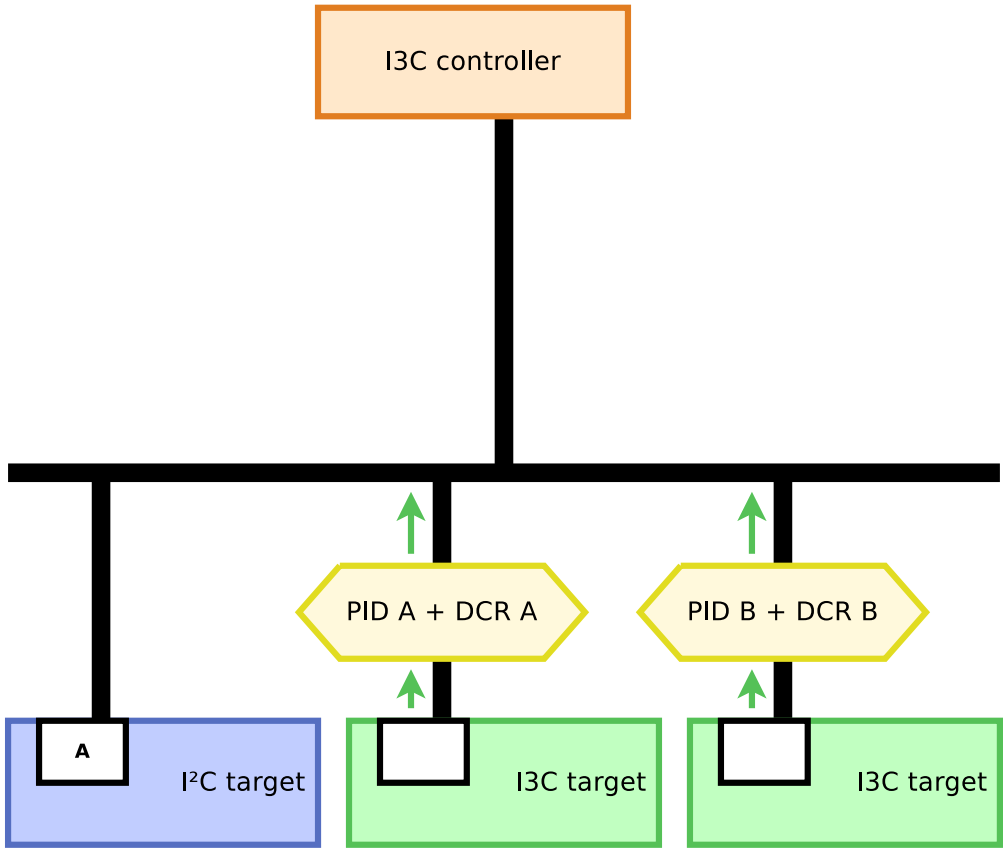


DAA

All I3C devices are expected to ACK.

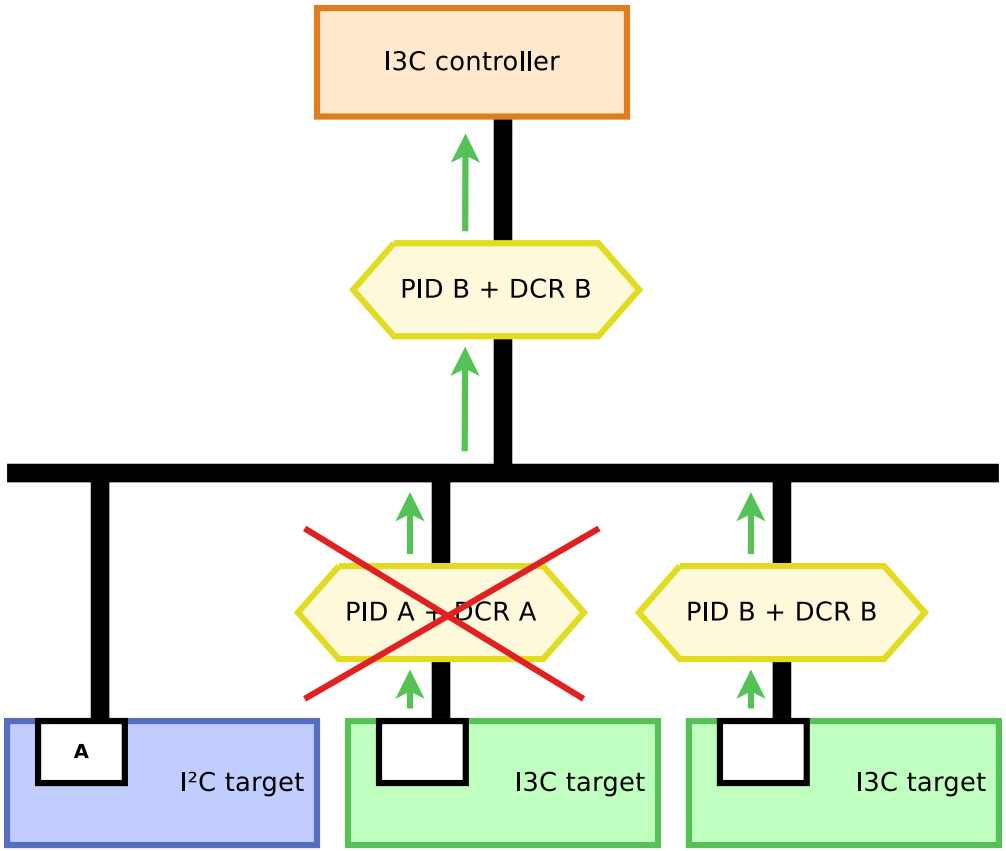


All I3C devices send their PID.



DAA

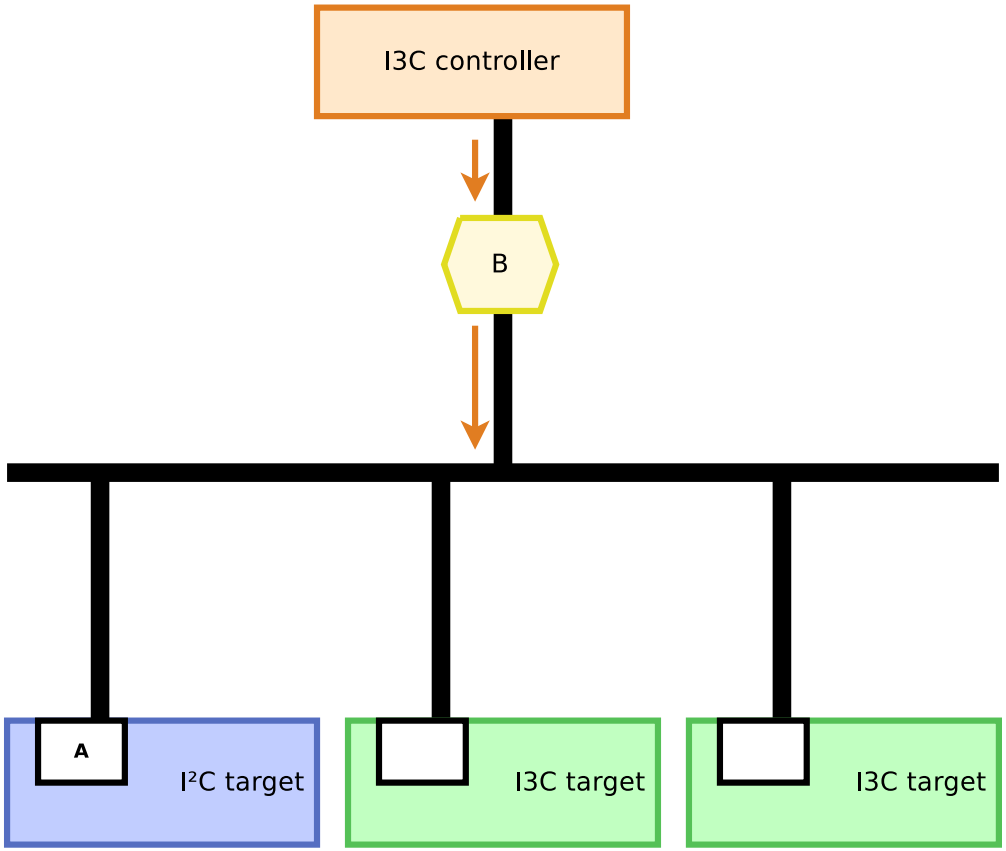
The target with the highest PID looses the arbitration (thanks to the open-drain nature of the SDA line during this exchange).



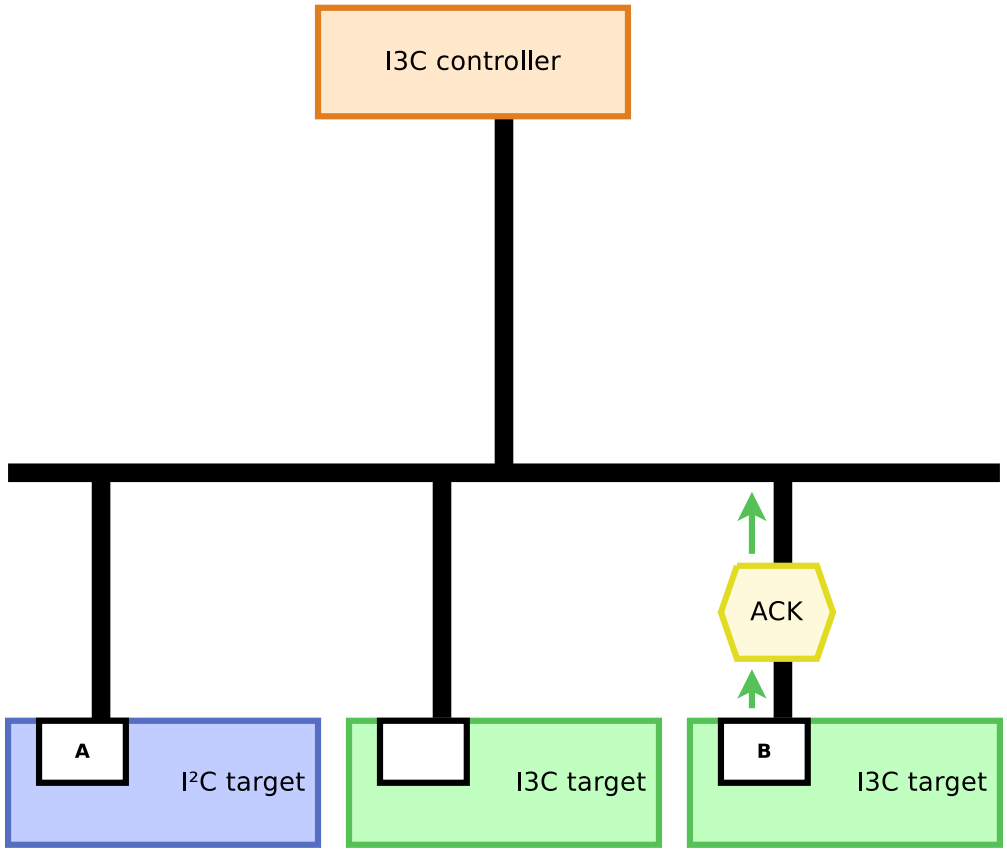
The winning device sends its PID followed by its BCR and DCR.

DAA

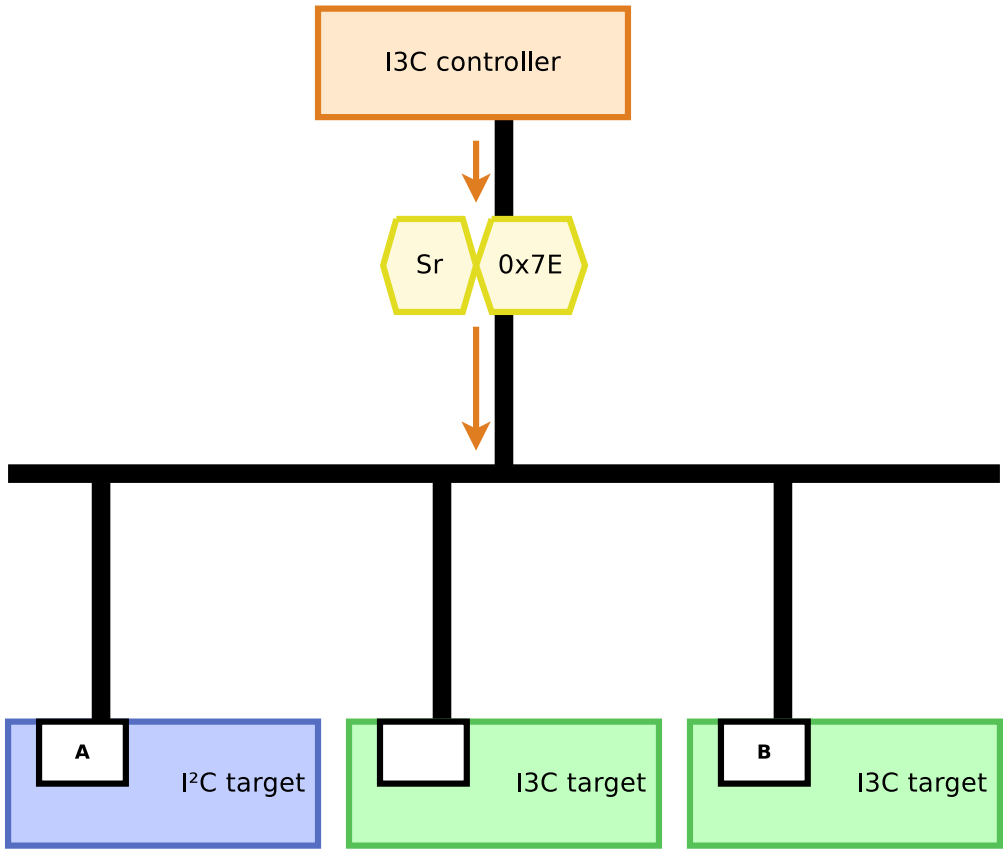
The controller sends a 7-bit address to the winning device that will become its own dynamic address.



The target will acknowledge this dynamic address.

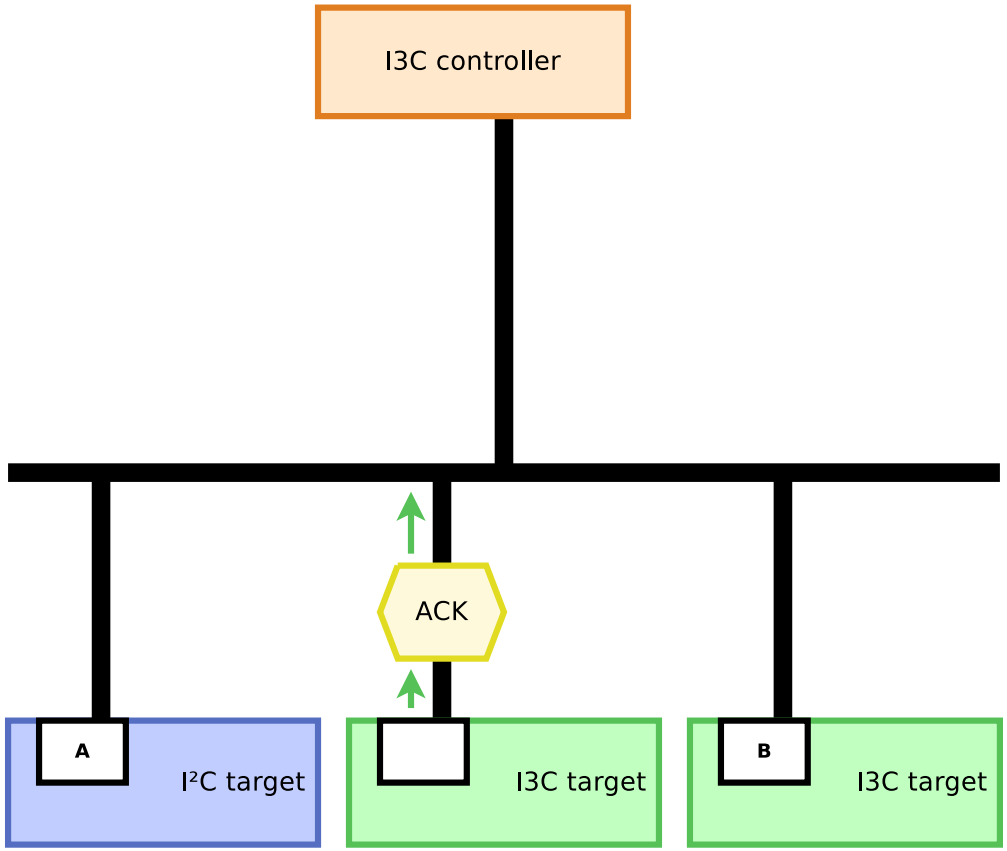


The controller generates a (Sr) condition and sends the broadcast address again.

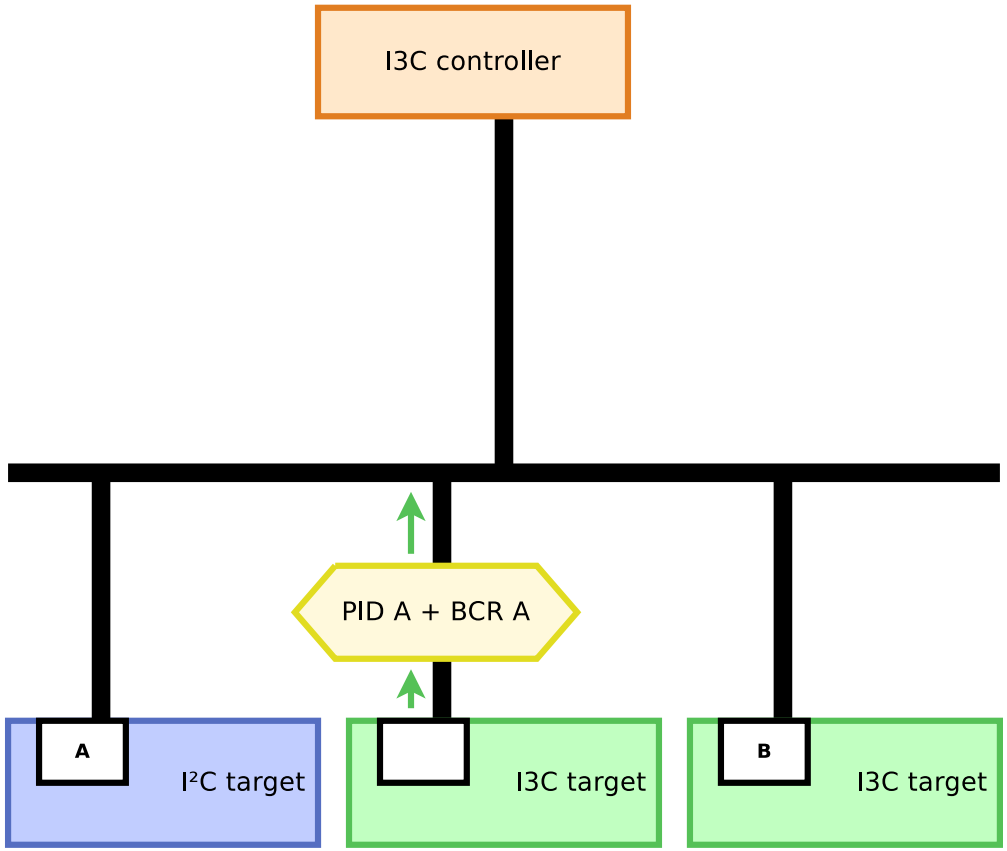


DAA

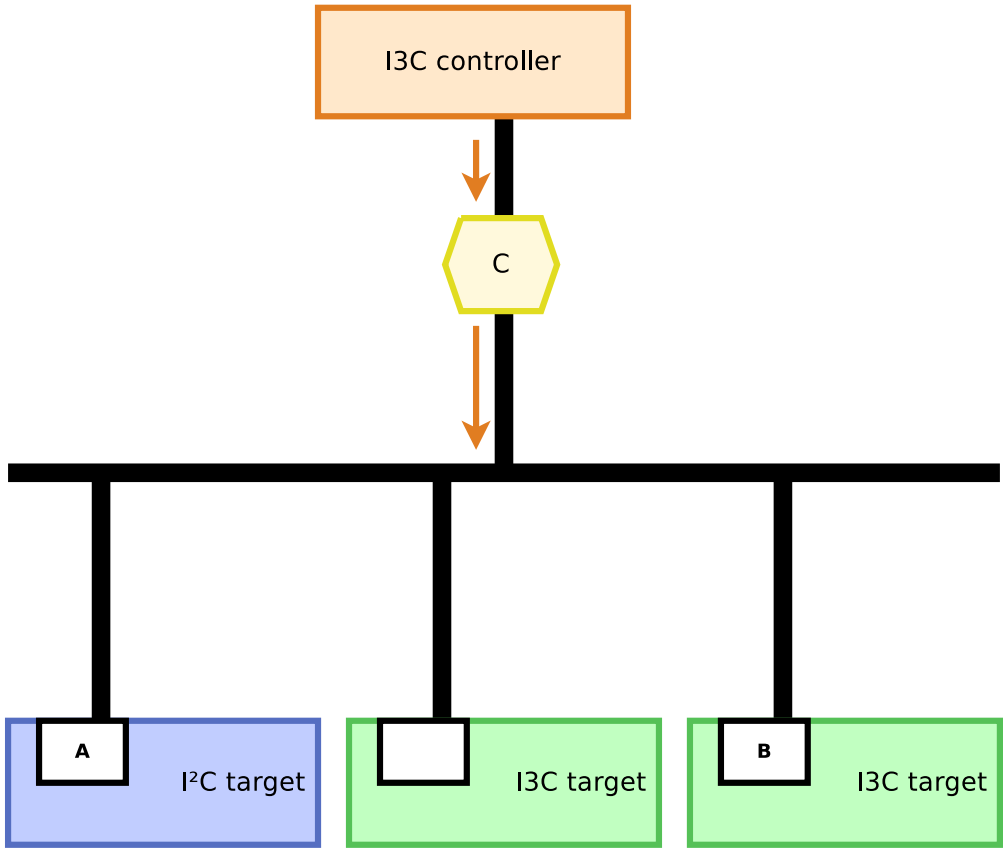
All devices that have not yet received a dynamic address should `ACK` it again.



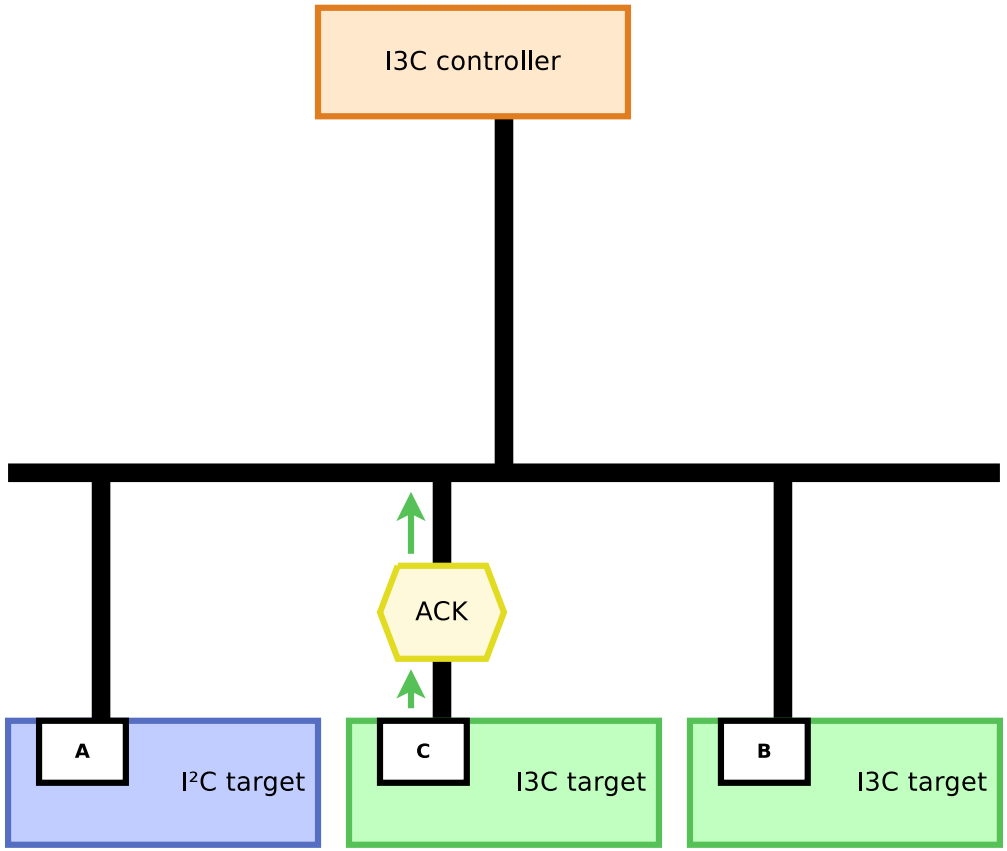
The new winning device sends its identification numbers.



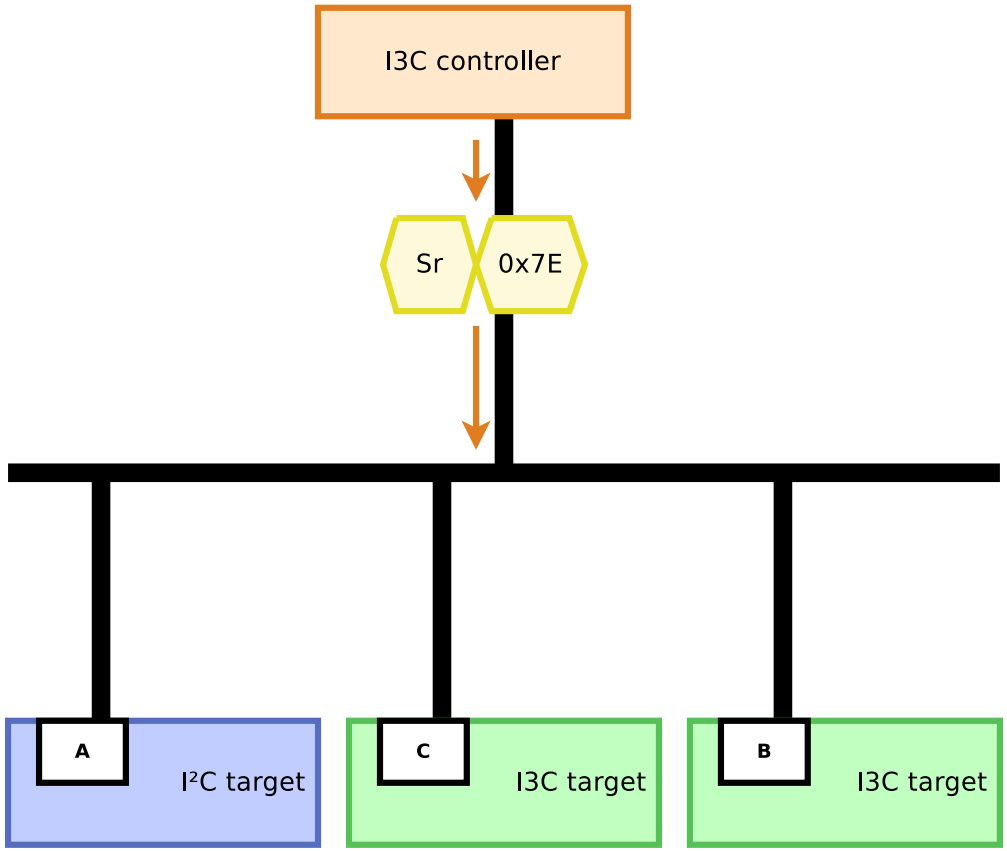
The controller provides to the device a different dynamic address.



The target device again acknowledges its new dynamic address.

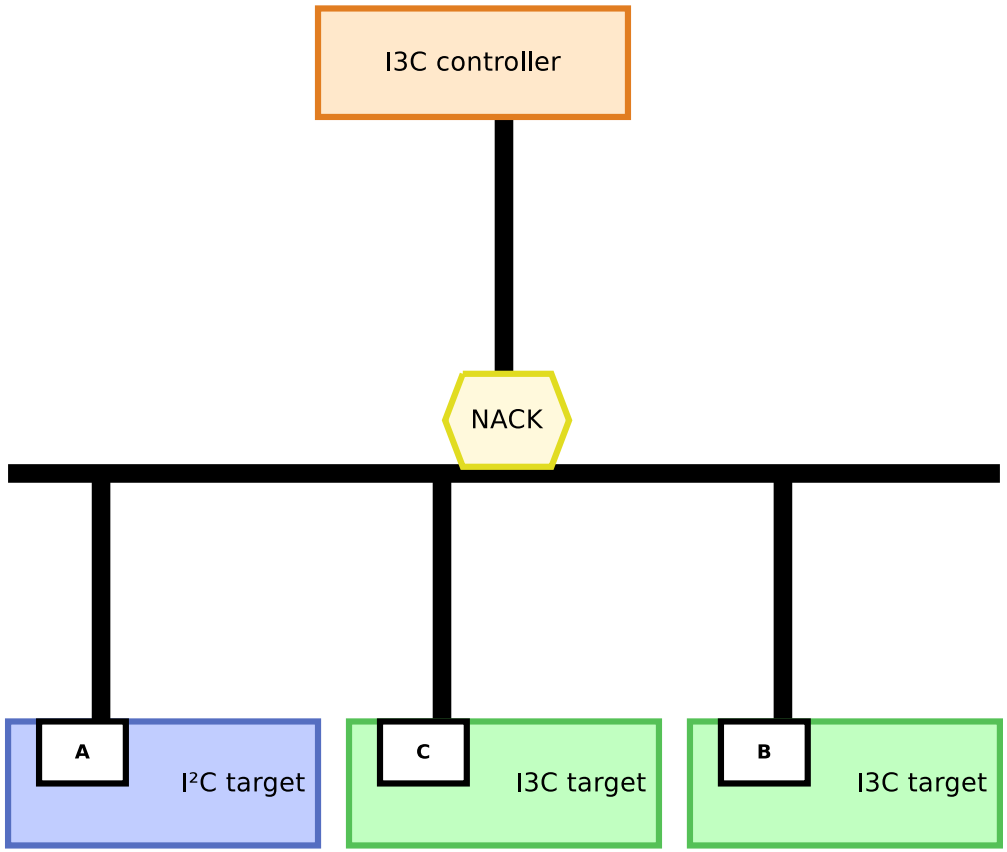


The controller keeps repeating the procedure...



DAA

...until there are no more devices acknowledging the request anymore, this means all woken-up devices have their dynamic address and the bus is now operational.



IBIs

- Device driven In-Band Interrupts
- The device can raise an interrupt in the following conditions:
  - The controller initiate a start condition: SDA low then SCL low
  - The bus is idle, the device asserts SDA low and waits for the controller to acknowledge the interrupt by pulling SCL low to end the start condition
- Device sends its address with a RnW bit to 1
- Priority arbitration depends on the dynamic address
  - Address exchange in open drain mode
    - 0 has a higher "priority" than 1
    - Lowest address ⇒ highest priority
- Controller ACKs or NACKs the interrupt
- Some devices may also provide a mandatory byte that the controller must read
- The controller decides to emit a Sr and continue talking on the bus or emits a P

Hot-join

- Devices can be unpowered and powered back at any moment, thus improving the energy savings
- Physically plugged-in devices at run-time
- A hot-join request is very much like an `IBI` but the device will provide a reserved address followed by a specific RnW bit to 0 in order to request a new `DAA` round

Controller handover

- Controller handover works very similarly than a hot-join, the device sending its own address this time, and keeping the RnW bit set to 0
- If the current controller does not feel ready, it may `NACK` the request
  - The offended target may request to be granted primary controller later
- If the controller agrees, it needs to prepare the bus for hand-off
  - Disable specific requests, wait for targets to finish their current processing, disable interrupts, ...
  - Assert a particular `CCC` to proceed the hand-off and end the transaction with a `P`
  - Arbitration has been lost, the controller should release the lines and check that the new controller now asserts its controllership

Time synchronization (I3C v1.1)

- A device might receive the order from the controller to generate samples at a given rate instead of on-demand only
- Controller would issue a `SETXTIME` to setup the frequency
- The target is then suppose to generate an `IBI` each time it triggers a sample

- Also a way of synchronizing different devices if an external low power clock is shared among several devices in order to trigger the sampling

Reset (I3C v1.1)

- Global reset:
  - Upon error showing that the bus is currently stuck, the I3C controller is capable of sending a global reset over the bus
  - Only supposed to reset the bus controller of each device
  - Implies restarting the DAA procedure

- Targeted reset:
  - A target might request to be reset
  - Saves an extra line on the final design
  - Involves asking all the devices but the one requesting the reset to discard the comming “reset pattern”

Reset (I3C v1.1)

- `RSTACT` specifies how the targets should react to a "reset pattern":
  - Discard
  - Reset the bus controller
  - Reset the entire chip
- A reset pattern is made of 14 `SDA` transitions with `SCL` kept low, a `Sr` and finally a `P`

Bulk Transport (I3C v1.1)

- New Bulk Transport (`HDR-BT`) mode for Bulk messages
  - Typical use: direct mapping of a SRAM
- Enhanced CRC checks with the sender knowing rapidly if the transfer must be repeated
- Lowered overhead
- The target can drive `SCL` during reads to avoid round-trip-time issues during long transfers

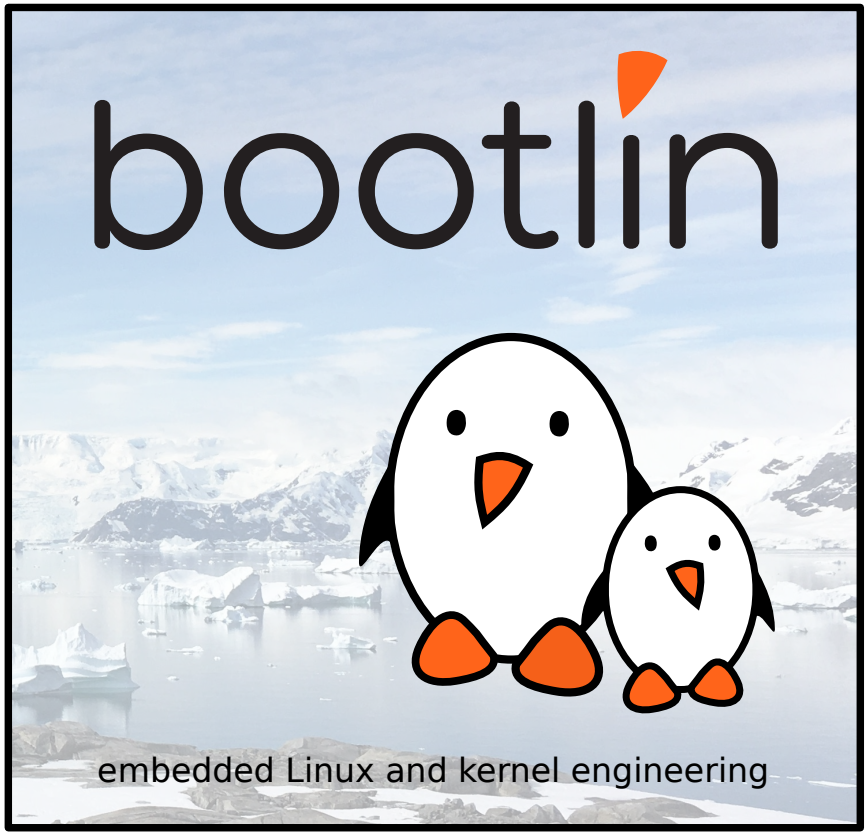
Multi-Lane (I3C v1.1)

- After so much efforts to keep everything in-band as much as possible, it was time to add new wires...
  - `SDR-ML DUAL`
  - `SDR-ML QUAD`
  - `HDR-DDR-ML DUAL`
  - `HDR-DDR-ML QUAD`
  - `HDR-TSP-ML DUAL`
  - `HDR-TSP-ML QUAD`
  - `HDR-BT-ML DUAL`
  - `HDR-BT-ML QUAD`

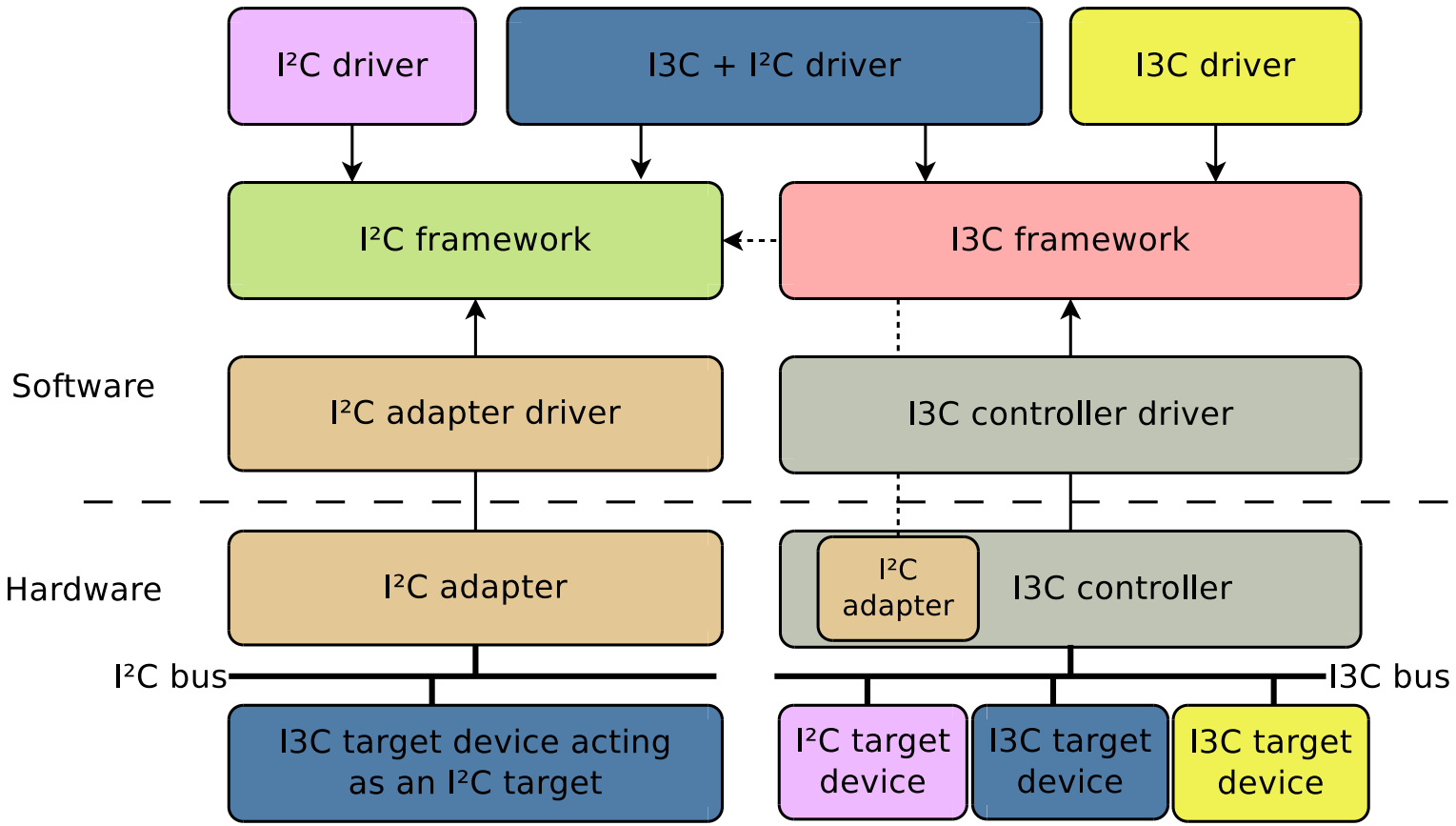
Linux API

Miquèl Raynal

miquel@bootlin.com



The I3C framework



Design choices

- Built as a separate subsystem than I2C in order not to break the I2C framework
- The subsystem defines:
  - A `struct i3c_bus` bus object
  - A `struct i3c_master_controller` controller structure
  - A rather exhaustive `struct i3c_master_controller_ops` interface
  - All the necessary APIs to register:
    - A controller
    - A device driver
    - `IBI`s
- Device-driver binding are based on the device's `PID`

The I3C bus infrastructure

include/linux/i3c/master.h

```
struct i3c_bus {
        struct i3c_dev_desc *cur_master;
        int id;
        unsigned long addrslots[((I2C_MAX_ADDR+1) * 2) / BITS_PER_LONG];
        enum i3c_bus_mode mode;
        struct {
                unsigned long i3c;
                unsigned long i2c;
        } scl_rate;
        struct {
                struct list_head i3c;
                struct list_head i2c;
        } devs;
        struct rw_semaphore lock;
};
```

The I3C controller structure

include/linux/i3c/master.h

```
struct i3c_master_controller {
        struct device dev;
        struct i3c_dev_desc *this;
        struct i2c_adapter i2c;
        const struct i3c_master_controller_ops *ops;
        unsigned int secondary : 1;
        unsigned int init_done : 1;
        struct {
                struct list_head i3c;
                struct list_head i2c;
        } boardinfo;
        struct i3c_bus bus;
        struct workqueue_struct *wq;
};
```

The I3C controller interface

include/linux/i3c/master.h

```
struct i3c_master_controller_ops {
        /* Bus management */
        int (*bus_init)(struct i3c_master_controller *master);
        void (*bus_cleanup)(struct i3c_master_controller *master);
        int (*do_daa)(struct i3c_master_controller *master);
        /* I3C devices management */
        int (*attach_i3c_dev)(struct i3c_dev_desc *dev);
        int (*reattach_i3c_dev)(struct i3c_dev_desc *dev, u8 old_dyn_addr);
        void (*detach_i3c_dev)(struct i3c_dev_desc *dev);
        /* CCCs transactions */
        bool (*supports_ccc_cmd)(struct i3c_master_controller *master,
                                 const struct i3c_ccc_cmd *cmd);
        int (*send_ccc_cmd)(struct i3c_master_controller *master, struct i3c_ccc_cmd *cmd);
        /* Private SDR transfers */
        int (*priv_xfers)(struct i3c_dev_desc *dev, struct i3c_priv_xfer *xfers, int nxfers);
        /* I2C devices management */
        int (*attach_i2c_dev)(struct i2c_dev_desc *dev);
        void (*detach_i2c_dev)(struct i2c_dev_desc *dev);
        int (*i2c_xfers)(struct i2c_dev_desc *dev, const struct i2c_msg *xfers, int nxfers);
        /* IBI management */
        int (*request_ibi)(struct i3c_dev_desc *dev, const struct i3c_ibi_setup *req);
        void (*free_ibi)(struct i3c_dev_desc *dev);
        int (*enable_ibi)(struct i3c_dev_desc *dev);
        int (*disable_ibi)(struct i3c_dev_desc *dev);
        void (*recycle_ibi_slot)(struct i3c_dev_desc *dev, struct i3c_ibi_slot *slot);
};
```

Additional information about bus initialization

- The core parses the information provided by the DT
- Static devices are instantiated on this base
- Calls the `->bus_init()` callback to initialize the controller
- Calls the `->do_daa()` callback to start the device discovery
  - Assigns dynamic addresses `DAA`
  - The way the controller wants to follow the `DAA` procedure is really open
  - For each device discovered, the controller driver calls `i3c_master_add_i3c_dev_locked()` to notify the core
- The core then registers the discovered devices to the device model

The device interface

include/linux/i3c/device.h

```
struct i3c_device {
        // Generic 'device-model' device structure
        struct device dev;
        // Internal representation of an I3C device
        struct i3c_dev_desc *desc;
        // Representation of the I3C bus, not tight to a specific
        // controller due to controller handover possibilities
        struct i3c_bus *bus;
};
```

An I3C device driver

Dummy I3C device driver

```
static int dummy_i3c_probe(struct i3c_device *i3cdev)
{
        ...
        i3cdev_set_drvdata(i3cdev, data);
}

static int dummy_i3c_remove(struct i3c_device *i3cdev)
{
        void *data = i3cdev_get_drvdata(i3cdev);
        ...
}

static const struct i3c_device_id dummy_i3cdev_ids[] = {
        I3C_DEVICE(<manufid>, <partid>, <driver-data>),
        { /* sentinel */ },
};

static struct i3c_driver dummy_i3c_drv = {
        .driver = {
                .name = "dummy-i3c",
        },
        .id_table = dummy_i3cdev_ids,
        .probe = dummy_i3c_probe,
        .remove = dummy_i3c_remove,
};
module_i3c_driver(dummy_i3c_drv);
```

Interacting with the I3C device

SDR transfers

```
u8 reg = 0x5, values[2];
struct i3c_priv_xfer xfers[2] = {
        {
                .flags = 0,
                .len = 1,
                .data.out = &reg,
        },
        {
                .flags = I3C_PRIV_XFER_READ,
                .len = 2,
                .data.in = values,
        },
};

ret = i3c_device_do_priv_xfers(i3cdev, xfers, ARRAY_SIZE(xfers));
if (ret)
        return ret;
```

⇒ CCCs for now are not exposed (subject to evolution)

Using IBIs

```
/* Called in a non-atomic context (workqueue) */
static void ibi_handler(struct i3c_device *i3cdev,
                        const struct i3c_ibi_payload *payload);

static int probe(struct i3c_device *i3cdev)
{
        struct i3c_ibi_setup ibireq = {
                .handler = ibi_handler,
                .max_payload_len = 2,
                .num_slots = 10,
        };

        ...
        ret = i3c_device_request_ibi(dev, &ibireq);
        if (ret)
                return ret;

        ret = i3c_device_enable_ibi(dev);
        if (ret)
                return ret;
        ...
}

static int remove(struct i3c_device *i3cdev)
{
        i3c_device_disable_ibi(i3cdev);
        i3c_device_free_ibi(i3cdev);
        ...
}
```

⇒ Slots are pre-allocated, possible overlap if not unqueued fast enough

Status and to-do list

- Things have moved forward
  - Hardware manufacturers more and more interested
  - Three controller drivers upstream
  - One IMU device driver upstream (not fully leveraging the power of I3C yet)
  - Ongoing controller handover work `https://lkml.kernel.org/lkml/1606716983-3645-1-git-send-email-pthombar@cadence.com/`
- There is still room for improvement
  - `HDR` support
  - I3C target interface
  - Global and directed resets
  - Time synchronization
  - `/dev` interface with additional user controls?
  - ...

Thanks!

Questions? Suggestions? Comments?

Miquèl Raynal

miquel@bootlin.com

https://bootlin.com/pub/conferences/2021/i3c/raynal-i3c