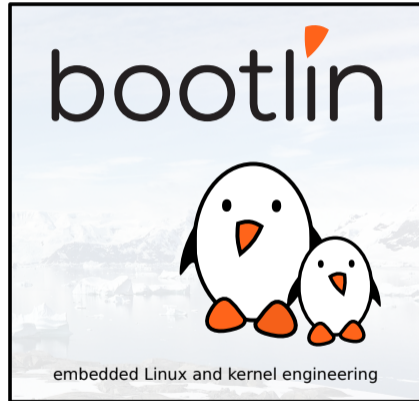




$1 + 1^2C = 13C$, what's
hiding in this additional
'1'

Boris Brezillon
boris@bootlin.com

© Copyright 2004-2018, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!





- ▶ Embedded Linux engineer and trainer at Bootlin
 - ▶ Embedded Linux **development**: kernel and driver development, system integration, boot time and power consumption optimization, consulting, etc.
 - ▶ Embedded Linux, Linux driver development, Yocto Project / OpenEmbedded and Buildroot **training courses**, with materials freely available under a Creative Commons license.
 - ▶ <https://bootlin.com>
- ▶ Contributions
 - ▶ **Maintainer of the MTD subsystem**
 - ▶ **Kernel support for various ARM SoCs**
 - ▶ **Submitted RFCs for an I3C subsystem**
- ▶ Living in **Toulouse**, south west of France



'I' like in ...

- ▶ I²C == Inter Integrated Circuit
- ▶ I3C == **Improved** Inter Integrated Circuit
- ▶ If it's improved, it's obviously better

Questions? Suggestions? Comments?

Boris Brezillon

`boris@bootlin.com`

Slides under CC-BY-SA 3.0

<https://bootlin.com/pub/conferences/2018/elc/bbrezillon-i3c/>

Backup slides

(for those who need more to be convinced)



$I + I^2C = I3C$, what's hiding in this additional 'I'

Introduction



I²C and SPI: fine but could do better

- ▶ I²C and SPI have long been the primary choice for embedded devices
 - + Both are relatively simple to implement (slaves and masters)
 - + Both require a limited amount of pins
 - I²C is quite slow
 - SPI requires one extra CS pin per device
 - Both need an extra pin per device to signal interrupts
 - Both don't support hotplug
 - Both don't support automated device discovery (though I²C bus can be scanned to know if a device is present at a specific address)
 - Address collisions on I²C buses



I3C: an attempt at addressing I²C/SPI limitations

- ▶ What if we could improve SPI and I²C to make something even more attractive
- ▶ Here comes I3C (MIPI standard):
 - + Only 2 pins required
 - + In Band Interrupts
 - + Higher throughput (up to 35Mb/s)
 - + Better energy efficiency than I²C on a per-transmitted-bit basis
 - + Supports hotplug
 - + Devices are discoverable
 - + Dynamic address assignment to avoid address collisions
 - + Backward compatible with I²C (to some extent)
 - + [Standardization of device profiles?]
 - Likely harder to implement



I + I²C = I3C, what's hiding in this additional 'I'

Few details about the protocol



I3C: backward compatibility with I²C

- ▶ I3C has been designed with I²C backward compatibility in mind
- ▶ Why?
- ▶ Because of the existing I²C slaves ecosystem (a huge variety of sensors and other kind of devices are controlled over I²C)
- ▶ Transition to systems containing only I3C sensors/slaves is likely to take long
- ▶ Some new/fancy sensors/slaves will support I3C natively
- ▶ But you'll still have to embed I²C slaves to have a full-blown system until I3C equivalences are available
- ▶ Backward compatibility is a two-way problem:
 - ▶ You'll have to connect I²C slaves on an I3C bus
 - ▶ Slave vendors might want to make their new I3C-compatible devices backward compatible with I²C so that can can indifferently be connected on an I3C or I²C bus

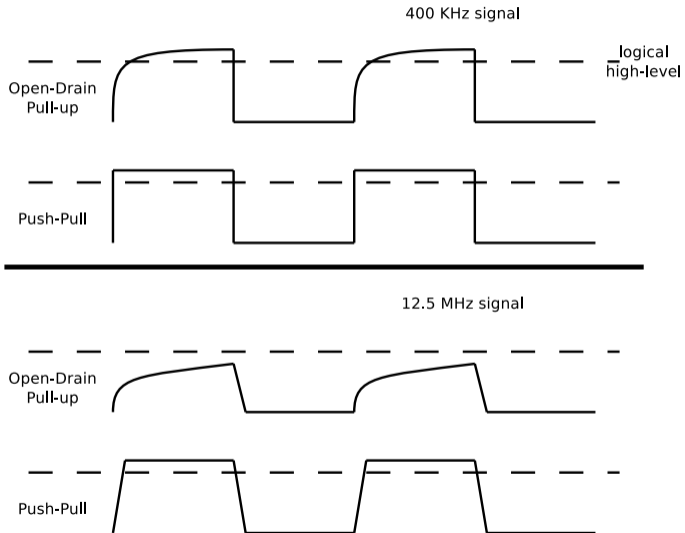


I3C vs I²C: physical layer

- ▶ Similarities:
 - ▶ Only 2 signals: SDA (data) and SCL (clock)
 - ▶ SDA in open-drain with a pull-up
 - ▶ Start/RepeatedStart/Stop/Ack/Nack conditions are unchanged
- ▶ What has changed:
 - ▶ SCL in push-pull →
 - ▶ Devices doing clock-stretching are not allowed
 - ▶ Clock frequency can be higher (due to faster rising edge)
 - ▶ SDA switches to push-pull when possible (to improve throughput and reduce power consumption)
 - ▶ HDR (High Data Rate) modes (DDR or Ternary based modes)



I³C vs I²C: why switching to push-pull?





I3C vs I²C: but I²C is open-drain...

- ▶ Why?
 - ▶ It's simpler to implement
 - ▶ Allows any device to easily take control of the bus:
 - ▶ Default state is high-level
 - ▶ The line is driven low if one or more devices set it low
 - ▶ No risk of having 2 devices driving the line at two different levels
- ▶ Remember that I3C wants to be backward compatible with I²C
 - ▶ SCL can easily be switched to push-pull if only one device (the master) drives it →
 - ▶ No clock-stretching
 - ▶ No I²C-like multi-master
 - ▶ SDA still needs to be controlled by slaves at specific time (to ACK/NACK a transaction)
 - ▶ I3C Master should dynamically switch from open-drain to push-pull during a transaction



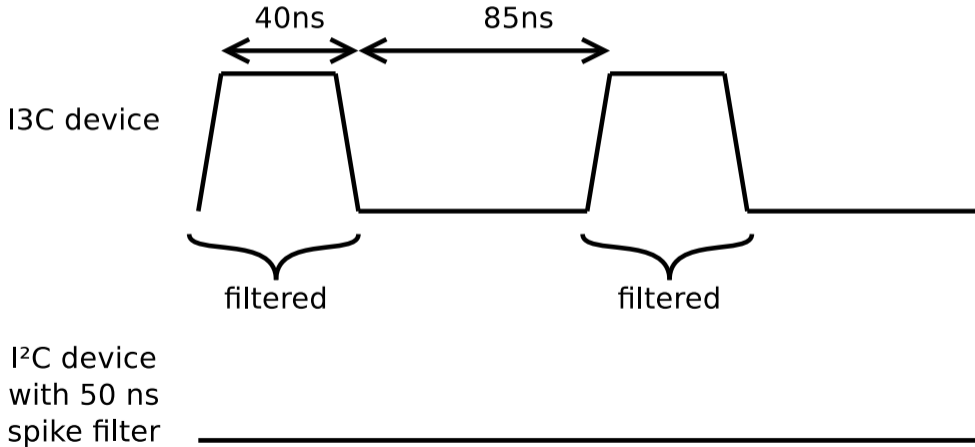
I3C: what about performance?

- ▶ SCL can run at up to 12.5 MHz in push-pull mode → 1.4 MByte/s
- ▶ Problem: I²C devices don't support such high-speed SCL
 - ▶ We need to slow down SCL when I²C devices are present on the bus...
 - ▶ ... unless we find a way to let them think the SCL signal stays low and exclude them from pure I3C transactions
 - ▶ Luckily, some devices have spike filters, filtering any changes that are maintained less than 50 ns
- ▶ I3C takes benefit of that by using an asymmetric SCL signal:
 - ▶ SCL stays high less than 50 ns (usually 40 ns since 12.5 MHz implies a 80 ns period)
 - ▶ The low period is extended (more than 50 ns) so that the I²C device always sees a logical low-level
- ▶ With this trick performance is still lower than when you have only I3C devices on the bus



I²C: 50 ns spike filter

8 MHz SCL signal





I3C: bus configurations

- ▶ Pure Bus: only I3C devices connected on the bus
 - ▶ SCL can be set to 12.5MHz
 - ▶ All HDR modes can be used
 - ▶ This is the ideal case in term of performance
- ▶ Mixed Fast Bus: I3C and I²C devices connected on the bus, but I²C devices have a 50 ns spike filter
 - ▶ SCL has to be lower (usually around 8.3MHz if you make low period twice as big as high period)
 - ▶ HDR-DDR and HDR-TSL can be used
 - ▶ This is a good compromise when you have to connect both I²C and I3C device on an I3C bus
- ▶ Mixed Slow Bus: I3C and I²C devices connected on the bus, but some I²C devices do not have a 50 ns spike filter
 - ▶ SCL is limited to the slowest I²C device on the bus
 - ▶ HDR modes are not supported



I3C vs I²C: addressing scheme

- ▶ Similarities:
 - ▶ 7-bit addresses
 - ▶ Devices are expected to ACK/NACK transactions
- ▶ What has changed:
 - ▶ Broadcast address (0x7e)
 - ▶ Can be used to address all I3C devices
 - ▶ Has been picked from the reserved address space of I²C to avoid collision with I²C devices
 - ▶ Addresses are no longer statically assigned: dynamically assigned by the master



I3C: I²C backward compatibility, let's sum-up

- ▶ Similarities in the PHY and MAC layer are motivated by backward compatibility
- ▶ Still, not all I²C slaves can be connected on an I3C bus along with I3C devices
 - ▶ Slaves doing clock-stretching are forbidden
 - ▶ Having a 50 ns spike filter on the I²C slaves end is recommended if you want to achieve acceptable performance on I3C transactions
 - ▶ Don't forget that some HDR modes can't be used when I²C devices are present on the bus



I3C vs I²C: functional differences

- ▶ This is probably the most interesting part for software developers
- ▶ I3C is functionally far from I²C:
 - ▶ Devices are assigned addresses by the master dynamically
 - ▶ Devices can be automatically discovered
 - ▶ Devices are self-descriptive (similar to USB devices)
 - ▶ Manufacturer and part id exposed
 - ▶ Concept of device class
 - ▶ Devices expose their bus-related capabilities
 - ▶ Some masters/slaves are hotplug-friendly
 - ▶ Bus management is more advanced/controlled than with I²C
 - ▶ Provides different types of transactions



I3C: CCC transactions or how to manage the bus

- ▶ I²C specification defines the bare minimum:
 - ▶ How to transmit things on the bus
 - ▶ How to interact with devices (address them and exchange data)
 - ▶ Lacks generic protocol to do bus management operations
- ▶ I3C protocol is more complex and I3C masters have a few more responsibilities:
 - ▶ Discover devices connected on the bus
 - ▶ Query information about those devices
 - ▶ Keep the bus in a consistent state
 - ▶ Do generic operations
- ▶ Requires standardization of bus management related operations
- ▶ This is done with CCC (Common Command Codes) transactions



I3C: CCC frame format

- ▶ An 8-bit opcode
 - ▶ Bit 7: 0 → broadcast, 1 → unicast
 - ▶ In case of unicast commands, the destination address is stored in the payload
- ▶ 0 to N bytes of payload
 - ▶ Opcode id defines whether the payload should be read or written
 - ▶ Payload length depends on the opcode
- ▶ A few examples:
 - ▶ ENTDAAs: 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)
 - ▶ ...

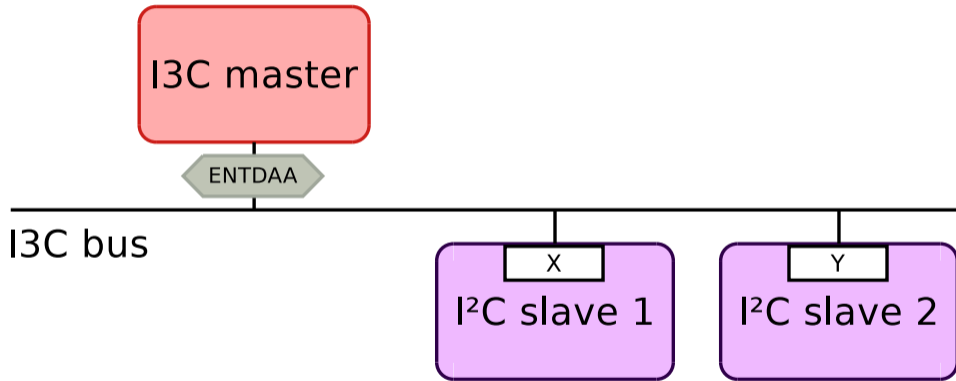


I3C: Discovering devices connected on the bus

- ▶ Discovery is done with the `ENTDAA` broadcast CCC command
- ▶ Every I3C slave device connected to the bus should reply to this command
- ▶ The Master follows the `ENTDAA` command by a `RepeatedStart` and waits for an `ACK` (which can be asserted by several devices since `SDA` is open drain in this situation)
- ▶ Every device connected on the bus should start emitting the following data:
 - ▶ `PID`: Unique ID containing a manufacturer ID, a part ID and an instance ID
 - ▶ `BCR`: The Bus Characteristics Register
 - ▶ `DCR`: The Device Characteristics Register
- ▶ While emitting, the slave should monitor the `SDA` state, and stop emitting as soon as `SDA` does not match (arbitration lost)
- ▶ Master will assign a dynamic address to the winning device
- ▶ The master will re-emit `RepeatedStart` until no-one `ACKs` the request, which means all devices have been discovered

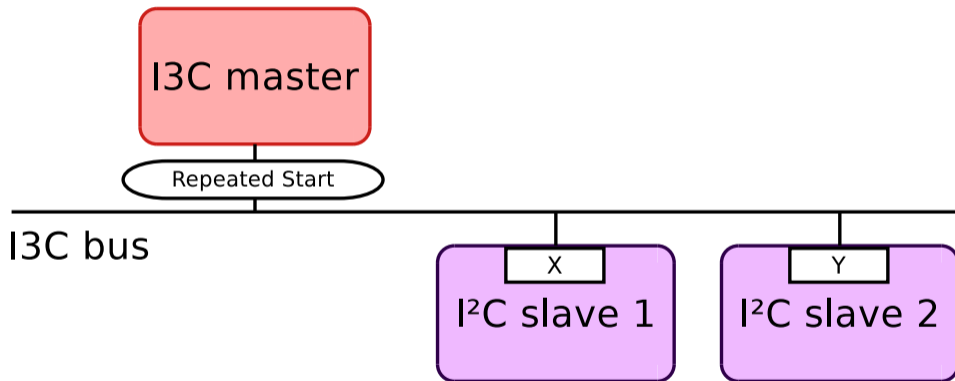


I3C: DAA (Dynamic Address Assignment)



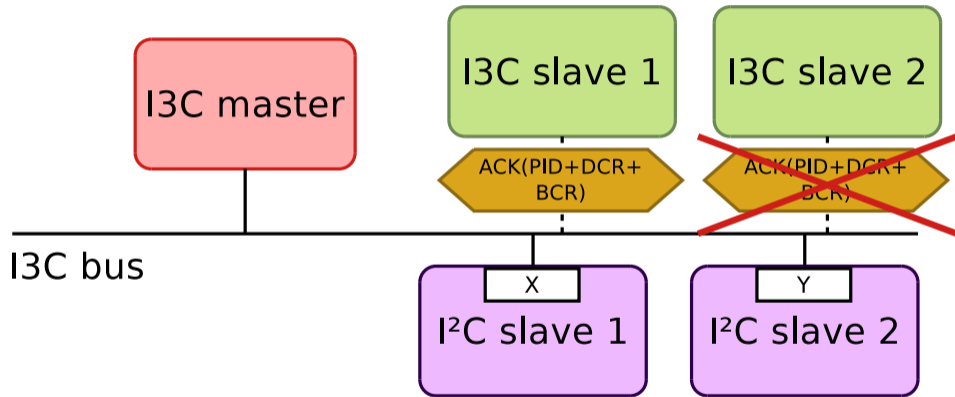


I3C: DAA (Dynamic Address Assignment)



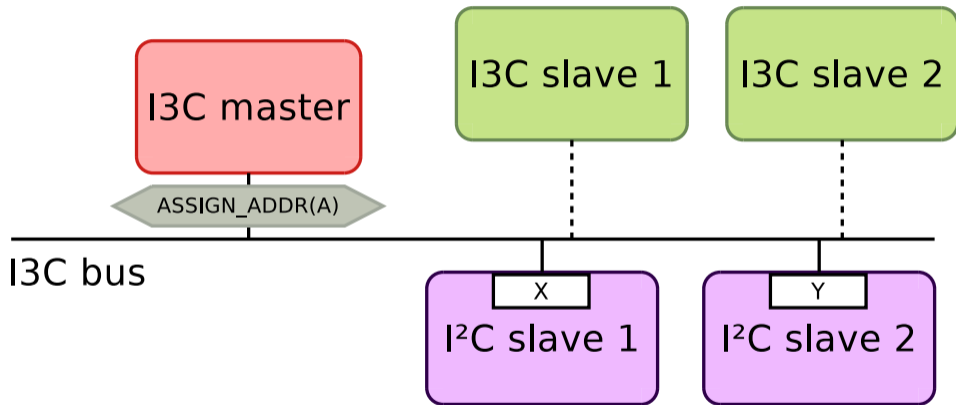


I3C: DAA (Dynamic Address Assignment)



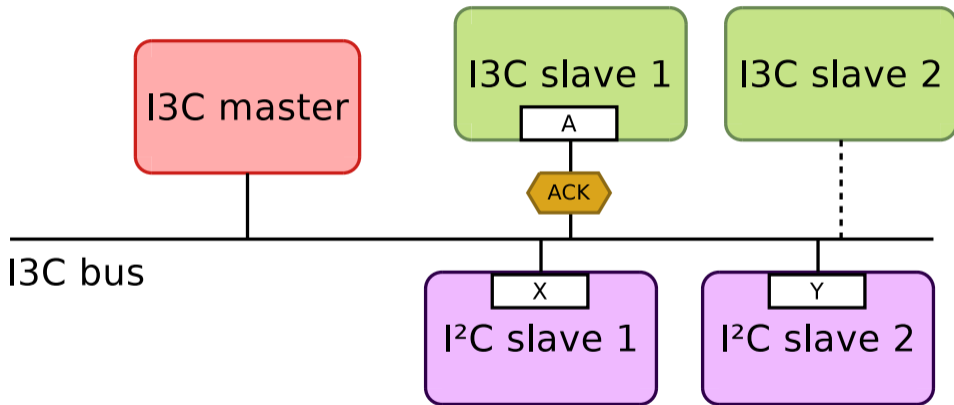


I3C: DAA (Dynamic Address Assignment)



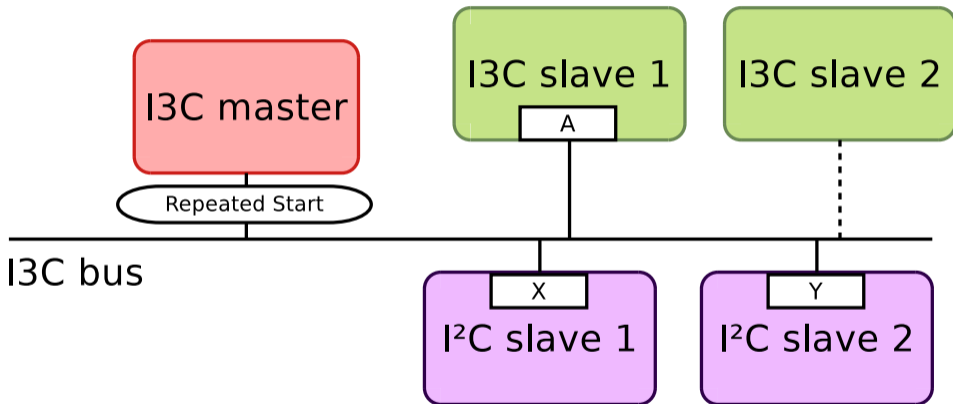


I3C: DAA (Dynamic Address Assignment)



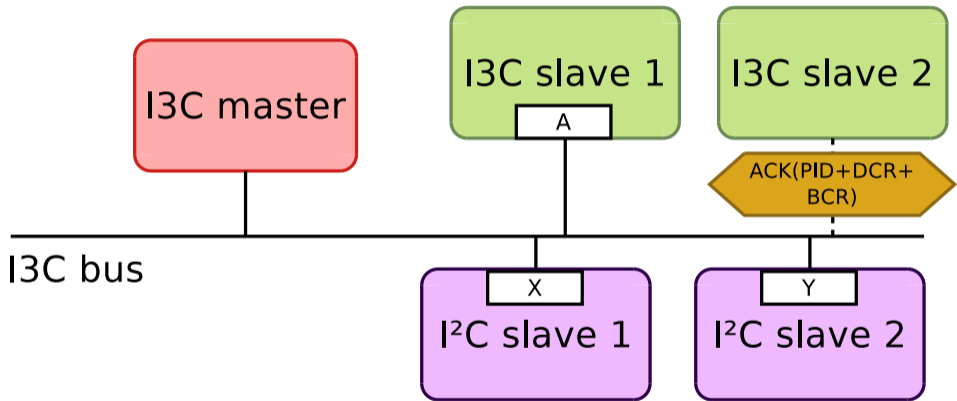


I3C: DAA (Dynamic Address Assignment)



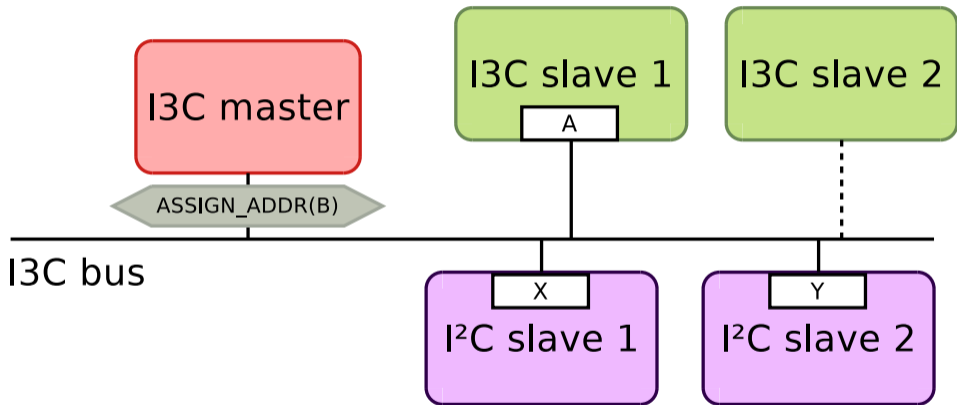


I3C: DAA (Dynamic Address Assignment)



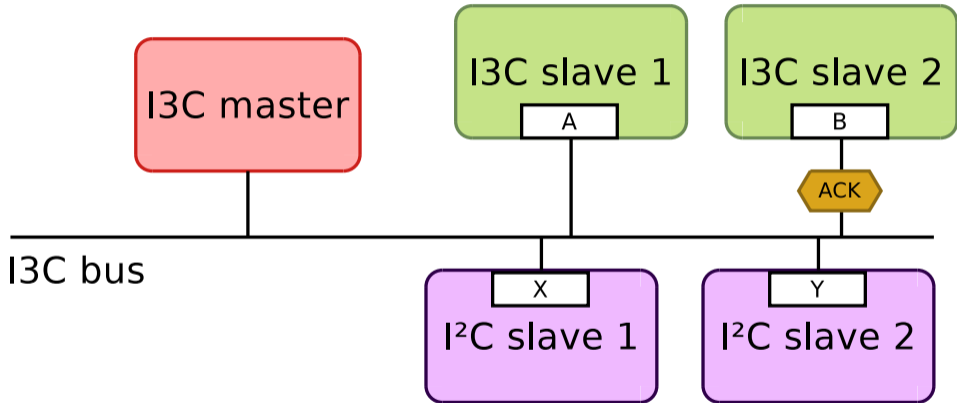


I3C: DAA (Dynamic Address Assignment)



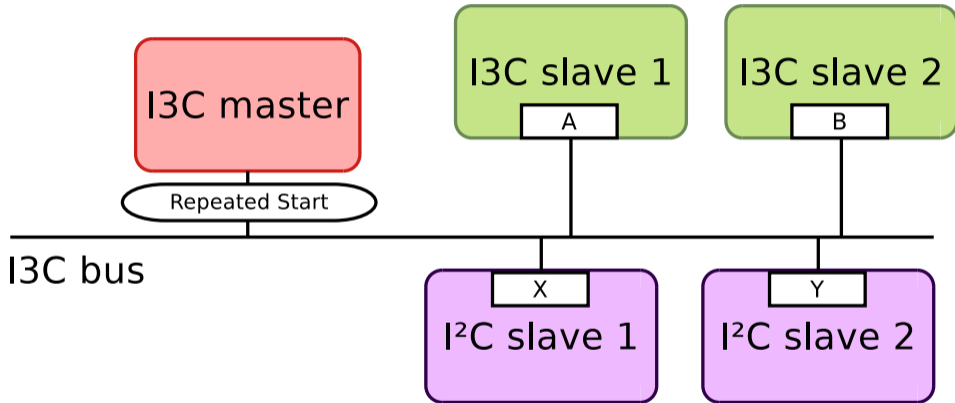


I3C: DAA (Dynamic Address Assignment)



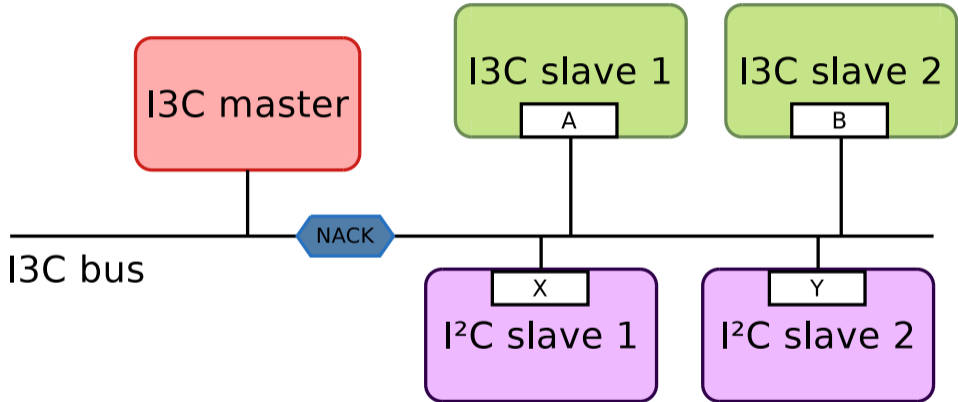


I3C: DAA (Dynamic Address Assignment)



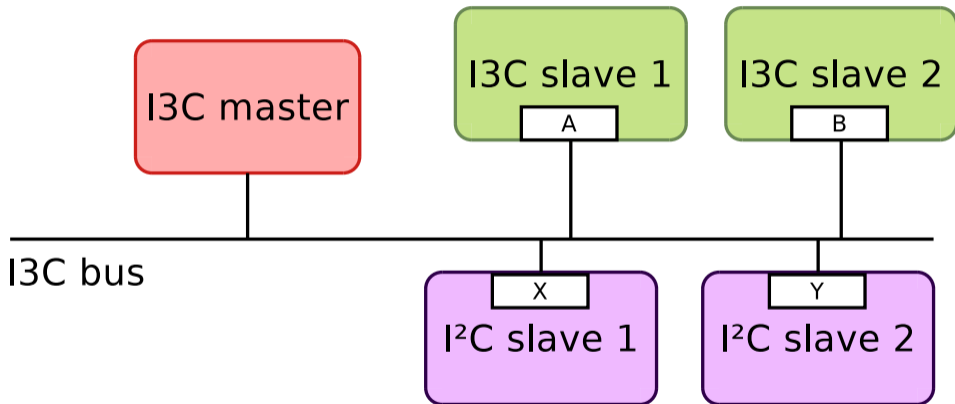


I3C: DAA (Dynamic Address Assignment)





I3C: DAA (Dynamic Address Assignment)





I3C: Identifying devices connected on the bus

- ▶ PID (Provisional ID): Unique ID formed of several subfields
 - ▶ Manufacturer ID: 15 bits describing the device manufacturer
 - ▶ Part ID: 16 bits describing the device
 - ▶ Instance ID: 4 bits in case you need to have several instances of the same device on a bus. Usually configurable through external pins
 - ▶ Extra information: 12 bits. Vendor specific. Still unclear what will be placed here
- ▶ BCR (Bus Characteristics Register): device capabilities
 - ▶ IBI capable
 - ▶ HDR capable
 - ▶ SDR transfer speed limitations
 - ▶ ...
- ▶ DCR (Device Characteristics Register): device type. IDs are standardized
 - ▶ IDs are standardized by MIPI
 - ▶ Will we have common interfaces standardized by MIPI??



I3C: SDR private transfers

- ▶ SDR transfers are supposed to replace I²C transfers for I3C devices
- ▶ Uses push-pull mode and higher SCL frequency (up to 12.5MHz) when applicable
- ▶ No standardization of the transfers content: devices are free to expose the interface they like



I3C: HDR transfers

- ▶ HDR (High Data Rate) transfers are intended to be used by devices that need high throughput
- ▶ 3 different HDR modes (all optional):
 - ▶ DDR (Double Data Rate): both edges of the clock are used to transmit data
 - ▶ TSP (Ternary Symbol Pure): both SCL and SDA are used to encode data. Only applicable to pure I3C buses
 - ▶ TSL (Ternary Symbol Legacy-inclusive-bus): same as TSP except it's applicable to buses on which you have I²C devices
- ▶ Based on 16-bit word transmission
- ▶ Frame format is defined by the spec → standard/generic HDR commands are likely to show up



I3C: IBIs or how to make your hardware designer happy

- ▶ IBI stands for In-Band Interrupts
- ▶ Removes the need for an extra pin to signal interrupts
- ▶ IBI generation is part of the protocol
 - ▶ Slave devices can preempt the bus to signal interrupts
 - ▶ Master is still in control and can `NACK IBIs`
 - ▶ `IBIs` can be followed by a payload
- ▶ Address arbitration takes place when several devices send IBIs concurrently
- ▶ The device with the lowest address wins



I3C: Hot-Join

- ▶ New name for hotplug, same feature
- ▶ I3C devices can signal their presence after the master has initialized the bus
- ▶ The master can `ACK` or `NACK` the request
- ▶ In case of `ACK`, the master should assign a dynamic address to the device by starting a `DAA` procedure



I3C: Multi master capabilities

- ▶ I3C natively supports multi-master
- ▶ Bus ownership is much more controlled than with I²C
- ▶ Gaining bus ownership involves a 2-way handshake between the currently active master and the one taking bus ownership
- ▶ 2 types of I3C masters:
 - ▶ Main master: the master responsible for initializing the bus
 - ▶ Secondary masters: masters that initially act as slaves and can at some point gain ownership of the bus
- ▶ There can only be one main master on a given I3C bus
- ▶ But you can have several secondary masters
- ▶ Bus ownership handover can be requested by:
 - ▶ an inactive master willing to control the bus
 - ▶ the active master if it doesn't want to act as a master anymore (for example, when being suspended)



I3C protocol: want to learn more?

- ▶ Very brief introduction to some of the I3C concepts
- ▶ Many pieces of information have been omitted
- ▶ If you want to know more, go check the I3C specification:
<http://resources.mipi.org/mipi-i3c-v1-download>



I + I²C = I3C, what's hiding in this additional 'I'

I3C support in Linux

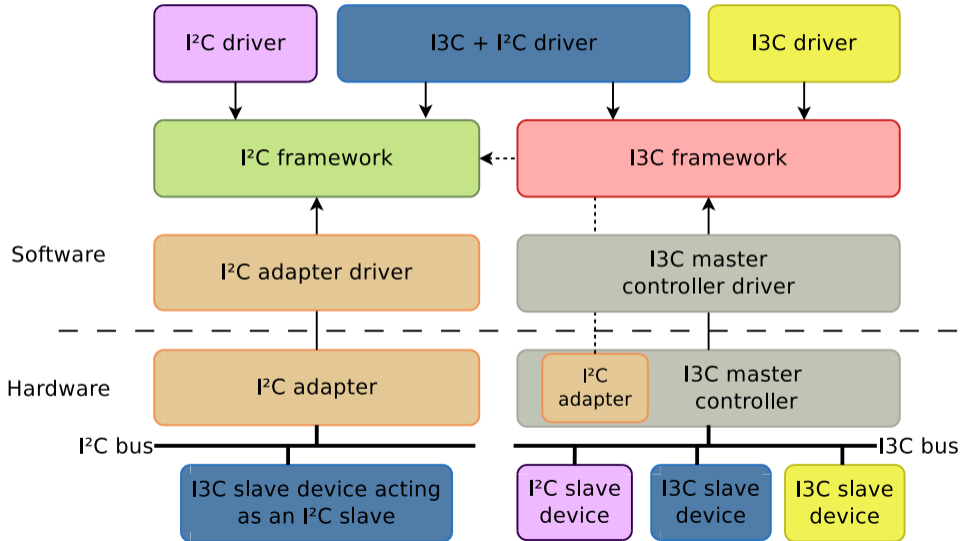


Supporting I3C in Linux

- ▶ I3C is a bit tricky to categorize
 - ▶ It is backward compatible with I²C devices
 - ▶ But I3C device handling is completely different
- ▶ Constraints:
 - ▶ We must keep the existing I²C ecosystem unchanged (all I²C drivers and users should work as before)
 - ▶ We should be able to support and expose I3C functionalities
- ▶ Two options:
 1. Extend the I²C framework to support I3C features
 2. Add an I3C framework and connect it to the I²C framework
- ▶ We went for option **2**



Linux I3C framework





Linux I3C framework: design choices

- ▶ Separate APIs for device drivers and I3C master controllers
- ▶ For each master controller an I3C and an I²C bus are created
- ▶ Both buses are connected through the controller parenting
- ▶ Device drivers are bound to devices based on the `DCR` or `PID` values
- ▶ IBIs are not exposed as regular IRQs (we tried this approach and it didn't fit well)



Linux I3C Device driver API: overview

- ▶ Exposed in **`include/linux/i3c/device.h`**
- ▶ Allows one to declare, register and unregister an I3C driver
- ▶ You can also register an hybrid I²C/I3C driver
- ▶ Provides a way to do SDR and HDR transfers (provided the device and master supports HDR modes)
- ▶ CCC commands are not exposed yet, since most of them are related to bus management (might change in the future if needed)
- ▶ Provides a way to register an IBI handler, and activate/deactivate the IBI



Linux I3C Device driver API: declaring a driver

```
static int dummy_i3c_probe(struct i3c_device *dev)
{
    ...
}

static int dummy_i3c_remove(struct i3c_device *dev)
{
    ...
}

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);
```



Linux I3C Device driver API: SDR private transfers

```
u8 reg = 0x5;
u8 values[2] = {0x1, 0x2};
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;

...
```




Linux I3C Device driver API: DDR transfers

```
#define MYVENDOR_READ_COMMAND      HDR_VENDOR_READ_CMD(0)
...

u16 data[4] = {};
struct i3c_hdr_cmd hdrcmd = {
    .mode = I3C_HDR_DDR,
    .code = MYVENDOR_READ_COMMAND,
    .ndatawords = ARRAY_SIZE(data),
    .data.in = data,
};

ret = i3c_device_send_hdr_cmds(i3cdev, &hdrcmd, 1);
if (ret)
    return ret;

...
```



Linux I3C Device driver API: IBIs

```
static void ibi_handler(struct i3c_device *dev,
                       const struct i3c_ibi_payload *payload)
{
    /* Called in a non-atomic context (workqueue) */
    ...
}

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);
    ...
}
```



Linux I3C Master controller driver API: overview

- ▶ Exposed in **include/linux/i3c/master.h**
- ▶ Allows one to register and unregister an I3C master controller
- ▶ Master controller drivers have to implement the `struct i3c_master_controller_ops` interface
- ▶ This interface tries to follow the I3C specification as much as possible
- ▶ Leaves a lot of freedom to drivers in how they deal with complex operations like DAA
- ▶ The framework provides generic helpers to help drivers implementing these operations



Linux I3C Master controller driver API: bus initialization

- ▶ One of the first things controllers are asked to handle is the bus initialization
- ▶ The core
 - ▶ parses information provided by the FW (currently, only DT parsing is supported)
 - ▶ instantiates I²C and I3C device objects based on this definition (those devices are not registered to the device model)
- ▶ And finally, the core calls the master controller `->bus_init()` method which is responsible for:
 - ▶ Configuring the controller to take bus limitations into account (based on defined I²C devices)
 - ▶ Pre-reserving dynamic addresses that are meant to be manually assigned before DAA
 - ▶ Doing DAA to discover devices connected to the bus
 - ▶ For each device discovered during DAA, the controller calls a core helper to add the new I3C device to the list of devices
- ▶ Only after all these steps, both I3C and I²C devices are registered to the device model



Linux I3C Master controller driver API: various kind of transfers

- ▶ One method for each:
 - ▶ `->priv_xfers()` is for private SDR transfers
 - ▶ `->send_hdr_cmds()` is for HDR transfers
 - ▶ `->send_ccc_cmd()` is for CCC transactions
 - ▶ `->i2c_xfers()` is for I²C transfers (used by the I²C → I3C glue)
- ▶ There's an extra `->supports_ccc_cmd()`, since not all CCC commands are mandatory
- ▶ All these methods are working in a synchronous manner
- ▶ `->priv_xfers()` and `->send_hdr_cmds()` can do several transfers in one go using RepeatedStart instead of Stop+Start



Linux I3C Master controller driver API: IBIs

- ▶ The master controller IBI interface follows the I3C device API:
 - ▶ `->request_ibi()`: allocate resources to later handle IBIs coming from a specific device
 - ▶ `->free_ibi()`: free resources allocated for IBIs coming from a specific device
 - ▶ `->enable_ibi()`: enable IBIs coming from a device
 - ▶ `->disable_ibi()`: disable IBIs coming from a device
 - ▶ `->recycle_ibi_slot()`: recycle a payload slot that was previously used to deliver an IBI to a device driver. Called after the IBI handler returns
- ▶ Design choices
 - ▶ IBIs are delivered to the device driver in a workqueue context. Simply because IBIs may lead to other transmissions on the bus, and the API does not allow that in an atomic-context
 - ▶ Still, you should refrain from doing everything from the IBI handler → it will prevent other IBIs from being delivered (maybe we should have a per-device workqueue...)
 - ▶ IBI slots are pre-allocated, so you may lose some IBIs if the device driver is not handling them fast enough



Linux I3C Master controller driver API: Hot-Join

- ▶ Master controllers are likely to generate an interrupt when a Hot-Join request is received
- ▶ Master controller drivers will have to schedule a work (using the IBI workqueue) to start DAA
- ▶ They can use helpers to declare new devices after DAA



I3C support in Linux: the status

- ▶ What has been implemented/tested?
 - ▶ All the APIs described in the previous slides
 - ▶ Master controller API has been tested with Cadence Master IP
 - ▶ All Slave APIs have been tested using a dummy driver interacting with a dummy slave IP provided by Cadence
 - ▶ Last version of the I3C patch series can be found here (reviews are welcome):
<https://lkml.org/lkml/2017/12/14/406>
- ▶ What's missing?
 - ▶ I3C Multi-master with bus ownership handover procedure
 - ▶ Slave controller API to support things like I3C gadget
 - ▶ HDR-TSP/TSL. Couldn't test it since Cadence master controller IP does not support these modes
 - ▶ We didn't have a real device driver, but this is being addressed (driver for an I3C gpio-expander will be part of the next version)

Questions?

Boris Brezillon

boris@bootlin.com

Slides under CC-BY-SA 3.0

<https://bootlin.com/pub/conferences/2018/elc/i3c/>

Support our crowdfunding campaign to develop
an upstream Linux kernel driver for Allwinner VPU

<https://bootlin.com/blog/allwinner-vpu-crowdfunding/>

