Zephyr® Project
Developer Summit

Zephyr I3C

Ryan McClelland - Meta

Brief I3C Overview

I3C / I2C New Terminology

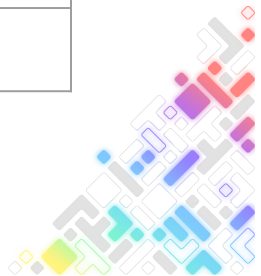
~~Master~~ = Controller

~~Slave~~ = Target

MIPI I3C specification v1.1.1
NXP I2C specification v7.0

I3C / I2C

	I2C	I3C
Bus Speed	100kHz, 400kHz, 1MHz	12.5MHz SDR (and HDRs)
Pull-up Resistors	External	Built-in (or External)
Physical Signaling	Open-Drain	Push-Pull and Open-Drain
Addressing	Static 7-bit/10-bit	Dynamic 7-bit
Interrupts	External I/O	In-Band
Hot Join	No	Yes
Common Command Codes	No	Yes



I3C Terminology

Primary Controller: the Controller-capable I3C Device that initializes the I3C Bus and performs configuration of all Target Devices. It acts as the authority for the Bus in its initial state, and becomes the first Active Controller once the Bus is configured.

Secondary Controller: A Controller-capable I3C Device that initially acts as a Target, but can also accept the Controller Role from any Active Controller (including the Primary Controller). Once a Secondary Controller accepts the Controller Role it becomes the new Active Controller, and may then pass the Controller Role along: either back to the previous Active Controller, or on to any other Controller-capable Device present on the I3C Bus.

Active Controller: The I3C Device that presently has control (i.e., the Controller Role) of the I3C Bus.

Target: A Target Device can only respond to either Common or individual commands from a Controller.

I3C Terminology

Dynamic Address (DA): A Device Address that is “assigned” during initialization of the Bus. Usually at Power On.

Static Address (SA): A Device Address that is fixed and cannot be changed. This is the same as the existing address that an I2C device uses. This an optional feature for I3C devices.

Group Address (GA): A Device Address that is “assigned” by the Active Controller. A Controller Device can send a given I3C Message to all Target Devices at once rather than one at a time with a Group Address. This is an optional feature for I3C devices.

Bus

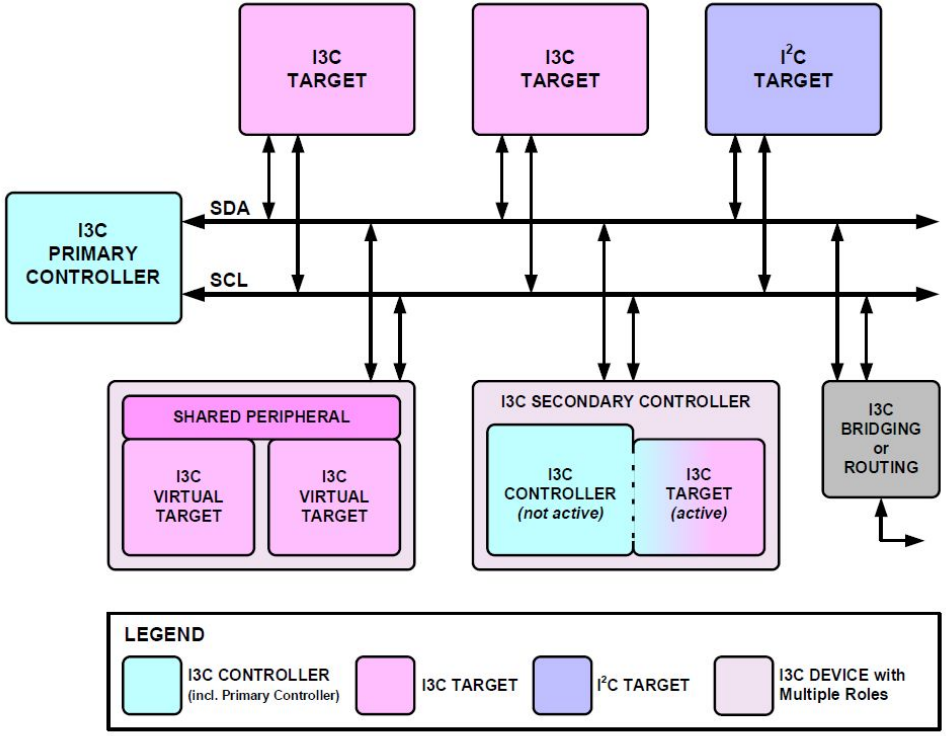


Figure 10 I3C Bus with I²C Devices and I3C Devices

I3C Common Command Codes (CCC)

Common Command Codes are an I3C standardized command set

Some are mandatory, conditional, or optional to be supported by controllers and targets

They can be broadcasted to all devices or directly to a single device

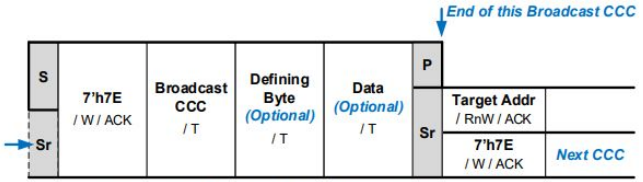


Figure 40 CCC Broadcast General Frame Format

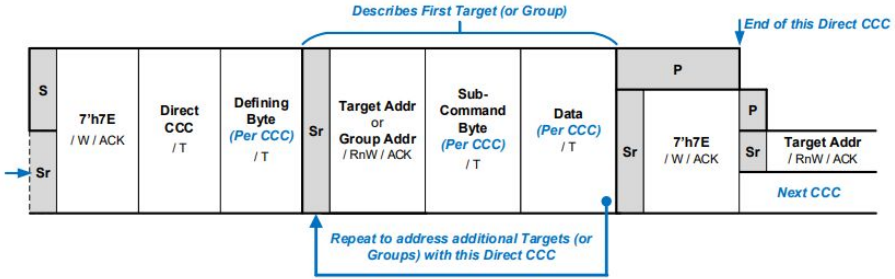


Figure 41 CCC Direct General Frame Format

I3C Provisional ID (PID)

Each I3C devices contains a Provisional ID (PID). This is a uniquely identifiable ID that is used to identify the device.

You will typically find this defined in the datasheet of the part.

The 48-bit Provisioned ID is composed of three parts:

1. **Bits[47:33]: MIPI Manufacturer ID *[MIPI01]*** (15 bits)

   Note: The Most Significant Bit of the MIPI Manufacturer ID is discarded, i.e. only the 15 Least Significant Bits are used.
2. **Bit[32]: Provisioned ID Type Selector** (One bit, 1'b1: Random Value, 1'b0: Vendor Fixed Value)
3. **Bits[31:0]:** 32 bits containing either a **Vendor Fixed Value** or a **Random Value**, depending on the value of Bit[32]:

   If the value of Bit[32] is 1'b0: Vendor Fixed Value, then:

   - **Bits[31:16]: Part ID:** The meaning of this 16-bit field is left to the Device vendor to define.
   - **Bits[15:12]: Instance ID:** The value in this 4-bit field should identify the individual Device, using a method selected by the system designer. For example: straps, fuses, non-volatile memory, or another appropriate method.
   - **Bits[11:0]:** The meaning of this 12-bit field is left for definition with additional meaning. For example: deeper Device Characteristics, which could optionally include Device Characteristic Register values.

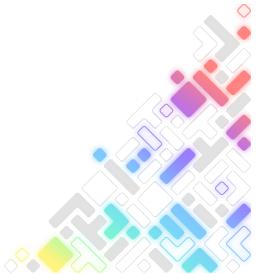
   If the value of Bit[32] is 1'b1: Random Value, then:

   - **Bits[31:0]:** 32-bit value randomly generated by the Device. This value can be queried via General Test Mode, using the Command Code **Enter Test Mode (ENTTM)** (see ***Section 5.1.9.3.8***).

I3C In-Band Interrupts

There are three types of IBIs

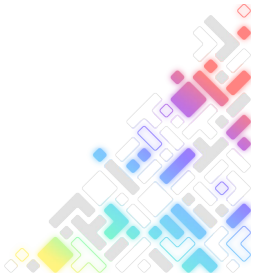
- **Hot-Join** - A Target is “Hot-Joining” an Active Bus. This is where the Target issues the IBI HJ command on the bus, and the Active Controller will assign it a Dynamic Address
- **Target Interrupt Request** - A Target is letting the Active Controller know that an “Event” happened. This can include data being ready from an IMU, replacing the need of using a GPIO interrupt.
- **Controller-Handoff** - A Secondary Controller is requesting to become the Active Controller. It is up to the Active Controller to “handoff” control of the bus to the Secondary Controller. It is still possible for an Active Controller to refuse a handoff.



I3C 9th Data Bit

With I2C, the 9th Data Bit was a ACK/NACK where for READS, it was used used to tell the target when the ‘read’ has ended. For WRITES, it was used for just for an acknowledgement of transfer.

With I3C, this 9th Data Bit is called the ‘T bit’. For READs, this gives the Target the control of when the transmission ends where it can issue a ‘End-of-Data’ (EoD) bit ending the transmission or a ‘not-End-of-Data’ (nEoD) bit to continue the transmission. The Controller can still issue an ‘Abort’ transfer in the middle of a transaction. For WRITEs, the ‘T bit’ is used as a Parity bit.



Zephyr I3C

I3C File List

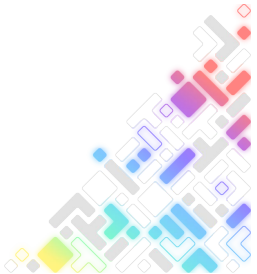
`include/zephyr/drivers/i3c/addresses.h` - I3C Address Related Helpers

`include/zephyr/drivers/i3c/ccc.h` - I3C CCC Helpers

`include/zephyr/drivers/i3c/devicetree.h` - I3C Devicetree Helpers

`include/zephyr/drivers/i3c/ibi.h` - I3C IBI Helpers

`include/zephyr/drivers/i3c/target_device.h` - I3C Target Device



I3C in Devicetree

For I3C devices, the ``reg`` property has 3 elements:

* The first one is the static address of the device.

  * Can be zero if static address is not used. Address will be assigned during DAA (Dynamic Address Assignment).

  * If non-zero and property ``assigned-address`` is not set, this will be the address of the device after SETDASA (Set Dynamic Address from Static Address) is issued.

* Second element is the upper 16-bit of the Provisioned ID (PID) which contains the manufacturer ID left-shifted by 1. This is the bits 33-47 (zero-based) of the 48-bit Provisioned ID.

* Third element contains the lower 32-bit of the Provisioned ID which is a combination of the part ID (left-shifted by 16, bits 16-31 of the PID) and the instance ID (left-shifted by 12, bits 12-15 of the PID).

```
i3c-dev0: i3c-dev0@420000ABCD12345678 {
        compatible = "vendor,i3c-dev";
        reg = < 0x42 0xABCD 0x12345678 >;
        status = "okay";
};
```

I3C-I2C in Devicetree

* The first reg entry is the static address of the device. This must be a valid address as I2C devices do not support dynamic address assignment.

* Second element is always zero.

  * This is used by various helper macros to determine whether the device tree entry corresponds to a I2C device.

* Third element is the LVR (Legacy Virtual Register):

```
i2c-dev0: i2c-dev0@380000000000000050 {
        compatible = "vendor,i2c-dev";
        reg = < 0x38 0x0 0x50 >;
        status = "okay";
};
```

I3C-I2C LVR

Table 7 Legacy I²C Virtual Register (LVR)

Bits	Name	Values
LVR[7:5]	**Legacy I²C-Only [2:0]** per ***Table 4***	**3'b000:** Index 0: Spike Filter, Max SCL clock freq tolerant N/A **3'b001:** Index 1: No Spike Filter, Max SCL clock freq tolerant **3'b010:** Index 2: No Spike Filter, Not Max SCL clock freq tolerant
		3'b011 – 3'b111: Index 3 – 7: Reserved
LVR[4]	**I²C Mode Indicator**	**0:** I²C Fm+ **1:** I²C Fm
LVR[3:0]	MIPI Alliance I3C WG Reserved	**0:** Reserved
		1 – 15: 15 available codes for describing the Device capabilities and function on the sensors' system.

I3C Device Descriptor Struct

Each Device is inserted in to a singly linked list that is contained within the Driver Data

Both the `device` struct for the “device” itself along with it’s `bus` device is stored

Contains all the required and optional I3C specification data for each device

```
struct i3c_device_desc {
    sys_snode_t node;
    const struct device * const bus;
    const struct device * const dev;
    const uint64_t pid:48;
    const uint8_t static_addr;
    const uint8_t init_dynamic_addr;
    uint8_t dynamic_addr;
    uint8_t group_addr;
    uint8_t bcr;
    uint8_t dcr;
    struct {
        uint8_t maxrd;
        uint8_t maxwr;
        uint32_t max_read_turnaround;
    } data_speed;
    struct {
        uint16_t mrl;
        uint16_t mwl;
        uint8_t max_ibi;
    } data_length;
    void *controller_priv;
    i3c_target_ibi_cb_t ibi_cb;
};
```

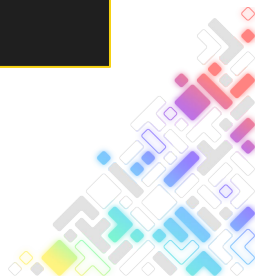
I3C APIs

The `struct i3c_device_desc` is looked up through the `i3c_device_find` function. Each bus will find the corresponding `i3c_device_desc` through the `i3c_device_id` which is the Provisional ID of the I3C device.

```
struct i3c_device_desc *i3c_device_find(const struct device *dev, const struct i3c_device_id *id)
```

The Provisional ID can be read from the devicetree.

```
struct i3c_device_id id = I3C_DEVICE_ID_DT(DT_NODELABEL(sensor));
```

I3C APIs

Unlike the I2C API, the ‘`bus`’ dev is not needed for private transfers. The `` `struct i3c_device_desc` `` is used instead as the `` `bus` `` dev is contained within it.

```
int i3c_transfer(struct i3c_device_desc *target, struct i3c_msg *msgs, uint8_t num_msgs);
```

However, I3C APIs that are ‘broadcasted’ still use the `` `struct device` `` pointer.

```
int i3c_do_ccc(const struct device *dev, struct i3c_ccc_payload *payload);
int i3c_do_daa(const struct device *dev)
```

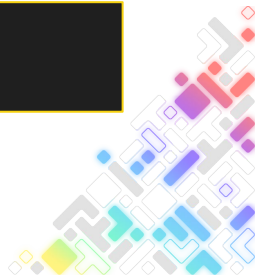
I3C IBI TIR

A Target can issue an In-Band Interrupt Target Interrupt Request (IBI TIR). When a Controller receives an IBI TIR for a target. It will enqueue the request (and payload) into a work queue.

```
int i3c_ibi_work_enqueue_target_irq(struct i3c_device_desc *target, uint8_t *payload, size_t payload_len);
```

It is up to the device driver for the target to set up the function pointer to the callback for when the IBI TIR is received.

```
struct i3c_device_desc *i3c_dev;
i3c_dev->ibi_cb = device_ibi_cb;
```

I3C IBI Raise

While acting as a Target, it is possible to issue an IBI to the Active Controller. These including issuing Hot-Join Request, Target Interrupt Requests, and Controller Handoff Request.

```
int i3c_ibi_raise(const struct device *dev, struct i3c_ibi *request)
```

```
struct i3c_ibi {
   /** Type of IBI. */
   enum i3c_ibi_type    ibi_type;

   /** Pointer to payload of IBI. */
   uint8_t          *payload;

   /** Length in bytes of the IBI payload. */
   uint8_t          payload_len;
};
```

I3C-I2C API

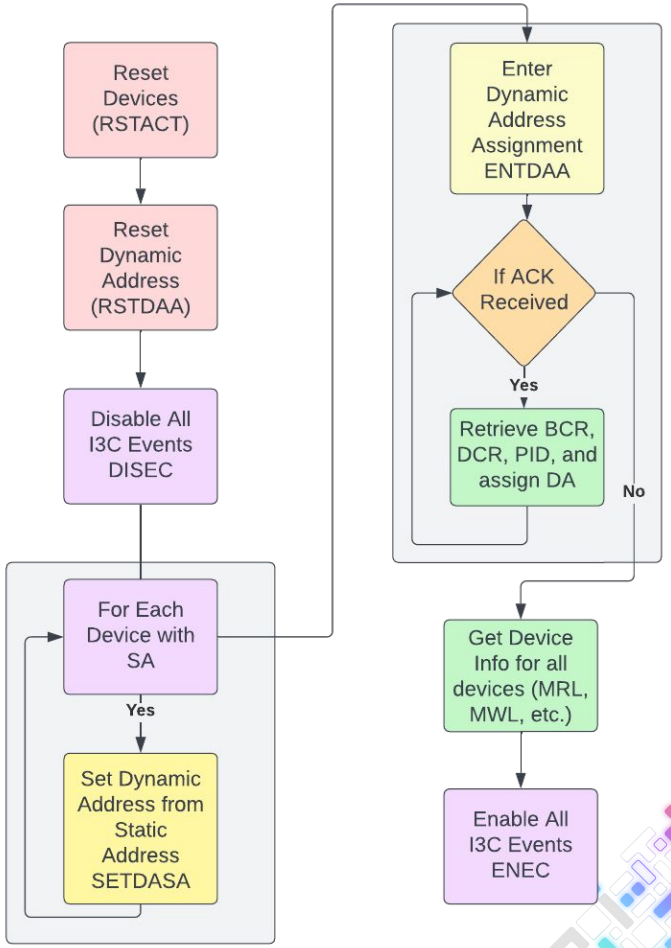
The I2C driver api is just ‘nested’ within the I3C driver api. This allows for all I2C apis to be used “as is”. This also means that all existing I2C device drivers such as a GPIO expander will just work without needing to have any I3C specific code.

```
__subsystem struct i3c_driver_api {
    /**
     * For backward compatibility to I2C API.
     *
     * @see i2c_driver_api for more information.
     *
     * (DO NOT MOVE! Must be at the beginning.)
     */
    struct i2c_driver_api i2c_api;
    ...
}

const struct device *i3c_dev = DEVICE_DT_GET(DT_NODELABEL(i3c));
i3c_transfer(i3c_dev,...);
i2c_transfer(i3c_dev,...);
```

I3C Bus Init

After a Primary Controller has finished initializing, it calls `i3c_bus_init` within i3c_common.c. This performs the necessary commands to initialize the I3C Bus with Dynamic Addresses along with gathering device info (BCR, DCR, PID, etc…)



Bus Characteristics Register (BCR

Table 5 Bus Characteristics Register (BCR)

Bit	Name	Description	Notes
BCR[7]	Device Role[1]	**2'b00:** I3C Target **2'b01:** I3C Controller-capable **2'b10:** Reserved for future definition by MIPI Alliance I3C WG **2'b11:** Reserved for future definition by MIPI Alliance I3C WG	1
BCR[6]	Device Role[0]		
BCR[5]	Advanced Capabilities	**1:** Supports optional advanced capabilities. Use GETCAPS CCC (***Section 5.1.9.3.19***) to determine which ones. **0:** Does not support optional advanced capabilities	2
BCR[4]	Virtual Target Support	**0:** Is not a Virtual Target and does not expose other downstream Device(s) **1:** Is a Virtual Target, or exposes other downstream Device(s)	3
BCR[3]	Offline Capable	**0:** Device will always respond to I3C Bus commands **1:** Device will not always respond to I3C Bus commands	4
BCR[2]	IBI Payload	**0:** No data bytes follow the accepted IBI **1:** One data byte (MDB) shall follow the accepted IBI, and additional data bytes may follow; see also the Set/Get Maximum Read Length CCC (***Section 5.1.9.3.6***). Data byte continuation is indicated by T-Bit per ***Section 5.1.2.3.4***. See also ***Section 5.1.8*** on use of IBI Payloads for Timing Control.	–
BCR[1]	IBI Request Capable	**0:** Not Capable **1:** Capable	–
BCR[0]	Max Data Speed Limitation	**0:** No Limitation **1:** Limitation	5

I3C Device Drivers

The device config is to contain the Provisional ID `i3c_device_id`, the `bus` device, and a pointer to the `i3c_device_desc`.

The device data is to contain the pointer to the `i3c_device_desc`.

When a device run its initialization function, it shall be expected to “find” it’s i3c device desc at runtime.

```
struct dev_config {
   struct i3c_device_desc **i3c;
   struct {
       const struct device *bus;
       const struct i3c_device_id dev_id;
   } i3c;
   ...
};

struct dev_data {
   struct i3c_device_desc *i3c_dev;
   ...
};

static int dev_init(const struct device* dev)
{
   const struct dev_config* const config = DEV_CFG(dev);

   struct dev_data* data = DEV_DATA(dev);
   data->i3c_dev = i3c_device_find(config->bus, &config->dev_id);
   if (data->i3c_dev == NULL) {
       LOG_ERR("Cannot find I3C device descriptor");
       return -ENODEV;
   }
   return 0;
}

.dev_cfg = {                    \
  .i3c = &dev_data_##inst.i3c_dev,        \
},                              \
.i3c.bus = DEVICE_DT_GET(DT_INST_BUS(inst)),        \
.i3c.dev_id = I3C_DEVICE_ID_DT_INST(inst),      \
```

Current Driver Support

SoC Support

NXP (Silvaco) I3C

Cadence I3C

NPCX - (Under Review)

Synopsys Designware I3C - Draft PRs

STM32H5 - No Driver

nRF54H20 - No Driver

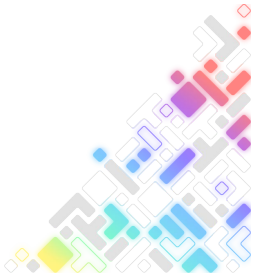
Device Support

LPS22HH

LPS2XDFW

stmemsc

BMI323, BMP581, ICM42688, ICM42670 - I3C interface not implemented

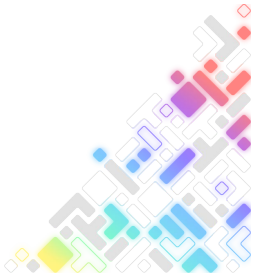


Future Zephyr I3C

I3C Shell

There are currently no Shell Commands to be used with I3C.
This can be useful to receive/send data over I3C in a test environment

https://github.com/zephyrproject-rtos/zephyr/issues/69941
https://github.com/zephyrproject-rtos/zephyr/pull/70773



I3C HDR-DDR Helper Macros, Parity, and CRC5

I3C HDR-DDR does contain a standard format for packets with Preamble, Tokens, Parity, Address, and CRC definitions. Macros are to be added within I3C Headers to be used by Drivers.

CRC5 function needed for the HDR-DDR CRC Word will be implemented within Zephyr's CRC subsystem

Even and Odd Parity Helper Functions are to also be added with in I3C Helper functions.

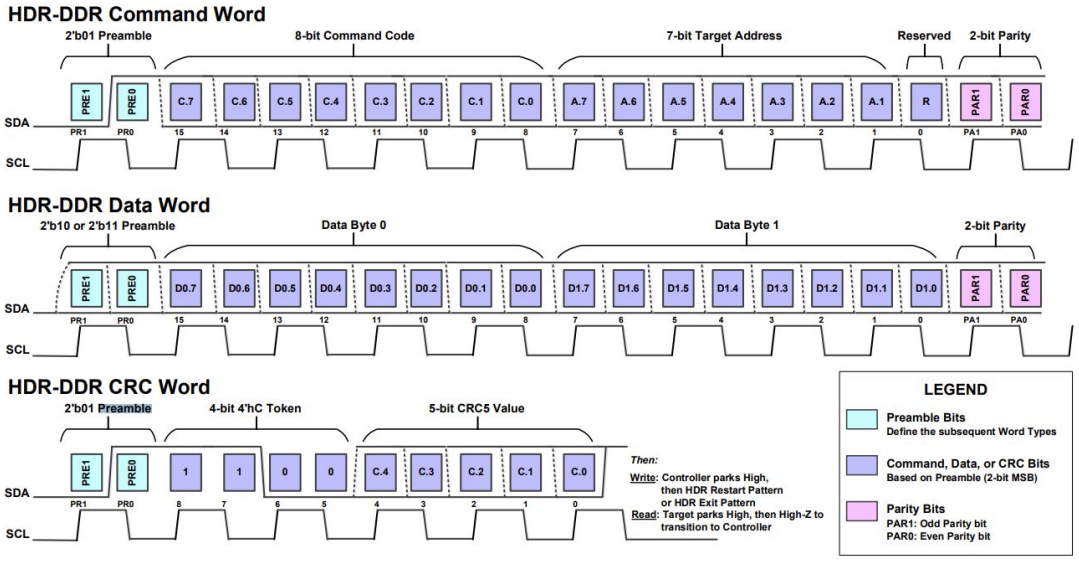
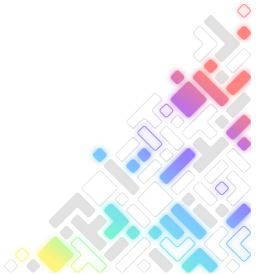


Figure 127 HDR-DDR Word Formats

I3C IBI Controller Handoff

Although there is an API in existence to raise an IBI Controller Handoff, there are currently no helper functions to implement this handoff between Controllers.

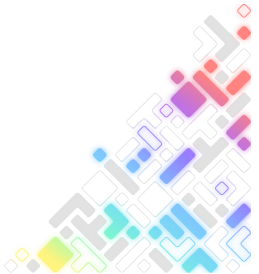
This would also require sending out the CCC DEFTGTS at the end of a bus initialization or if a Secondary Controller is discovered Hot-Join'ing the bus. This would inform all Secondary Controllers of all the devices on the Bus with their current Dynamic Address, PID, BCR, and DCR as well as I2C devices along with it's LVR.



I3C Timing Control

Certain Devices support I3C Timing Control. These would include the sample time of an event, such as a gyroscope/accelerometer value from an IMU, through the IBI TIR event such as through the I3C Async Mode 0.

This would require the Controller reading the clock frequency used by the Target with the CCC GETXTIME to correlate the sample time ticks to it’s own clock frequency in order for it to calculate the actual sample time of the event.



I3C-I2C Routing/Mux Devices

Currently, attaching I3C or I2C devices that are on muxes is not easily supported. This currently involves some devicetree ‘hacks’ or some odd attachments at runtime in order to make it “just work”.

An issue with this is that each route down the mux/router could have it’s own set of dynamic and static addresses. Currently Zephyr I3C Dynamic Address Assignment does not take in to account there may be devices with the same dynamic or static address just multiplexed to the same bus controller.

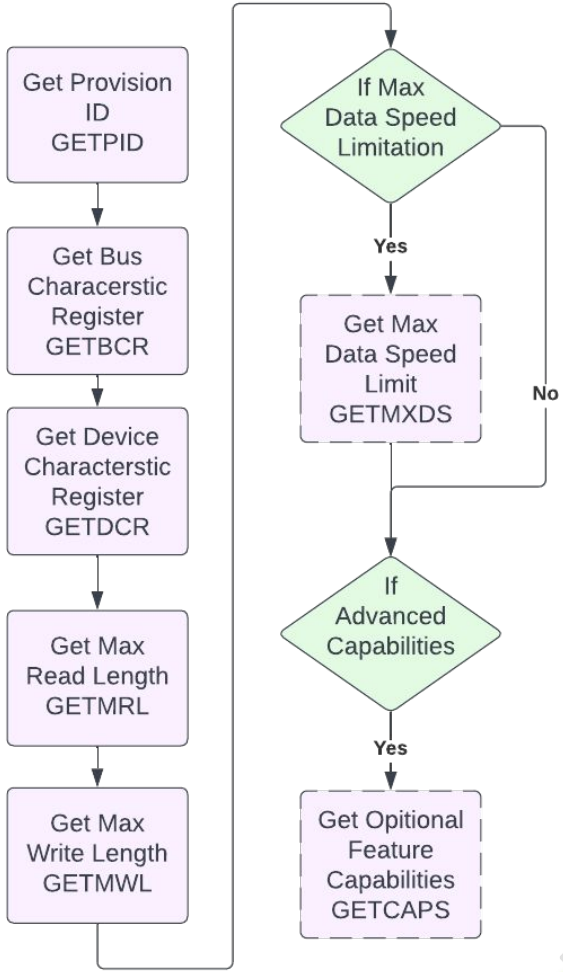
Current Devicetree reader macros do not read past the first level of devices. This is problematic as I3C/I2C muxes are under devicetree children of existing devices.

https://github.com/zephyrproject-rtos/zephyr/issues/61928

I3C Get Device Info

Currently the common `i3c_get_device_info` function doesn't do a complete read of all the device info.

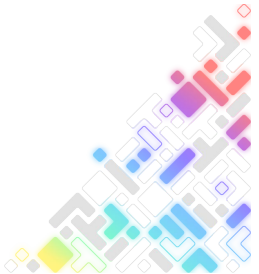
Depending on bits within the BCR, there is more information to be read about the device such as advanced capabilities and max data speed limit.



USB-I3C Device Class

In 2022, the USB Implementers Foundation unveiled the USB I3C Device Class. This would allow for an interface to expose and configure the I3C Function within a USB Device to allow interaction between Host software and the I3C Device, to drive transactions on I3C Bus to/from Target devices.

https://www.usb.org/document-library/usb-i3cr-device-class-specification-v11



Questions?

Zephyr® Project

Developer Summit