I3C Autonomous Slave Arch and µ-Arch spec

Overview of details for Autonomous (no Software) Slave RTL

Revision history

Revision	Date	Description	Author
0.1b	11/13/16 06/02/17	1st draft – requirements and arch. Added 1st info on magic regs.	PK

Contents

Figures

1. Introduction

MIPI I3C is a follow on to i2c which has major improvements in use and power, as well as providing an alternative to SPI for mid-speed. In particular:

- 2 wire multi-drop bus capable of 12MHz clock speeds with up to 11 devices
    - While using standard pads (vs. i2c special pads) with 4mA drive
    - Slave addresses are dynamically assigned – does not require a static address
        - But, slaves **may** have a static address at start
    - Slaves normally use inbound clock as the peripheral clock
        - So devices may have slow/inaccurate clocks internally
    - For read from Slave, Slave normally ends the read, but Master may terminate
        - Unlike i2c and SPI with the problems of Master having to “know” length
- In-Band interrupts, allowing Slaves to notify Master
    - Is equivalent to a separate GPIO, but can also be directly data bearing with a value
    - Prioritized so that if multiple Slaves wish to interrupt at the same time, the order is resolved
        - Dynamic addresses used for this, so controlled by Master
    - Interrupts can be started even when Master is not active on the bus, and yet no free running clock needed
    - Time-stamping option to allow resolution of initial event vs. when interrupt gets through
- Built-in Commands in separate “space” to not collide with normal Master->Slave messages
    - Controls bus behavior, modes and states, low power state, enquiries, etc.
    - Has additional room for new built-in commands to be used by other groups
- Hot-join onto bus allows devices to come on-line later than initial bus bringup
    - May be due to late wake up (power up) or physical insertion
        - Leaves physical insertion handling to system
    - Provides clean method for notification.
- Mixed i2c and i3c capable
    - I3C has support for certain legacy i2c devices on the bus
        - They must have spike filter and may not stretch the clock
    - I3C Slave devices such as this capable of operating on i2c buses
    - Also support for bridging (to i2c, SPI, UART, etc)

The I3C Autonomous block supports the base feature set, and uses parameterization to allow reduction of the logic to what is needed.
Autonomous means that the block is standalone. Its use is for ASICs and other State-machine oriented devices. For processor based devices, see the normal I3C Peripheral spec. It allows a Software interface.
The autonomous model works on the principle of a parameter driven generation of 8-bit “registers” which various properties to support the normal i2c and I3C model of indexed write and read.
Figure 1 shows the normal direct mapping which allows indexed registers and also controls “runs” which will be sequences for reads and max direct writes. The registers can be accessed in a group (continue a read or write), or may be individually indexed and so accessed. In the figure, index 9 is empty, so cannot be accessed.

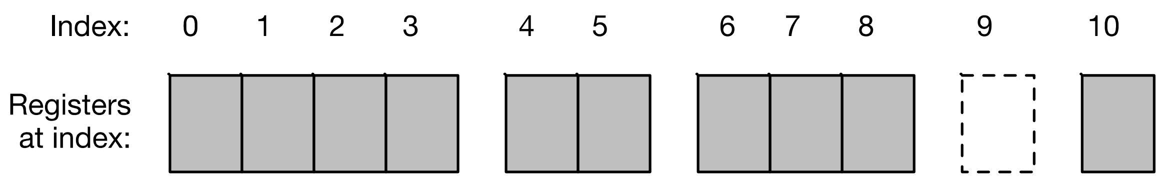

Figure 1 - showing normal direct mapped registers. Gaps show breaks in runs

Figure 2 shows a remapped use of registers which allows the index to only access from the 1st of a run of registers and the continuation of the read or write will access each sub-register in the run.

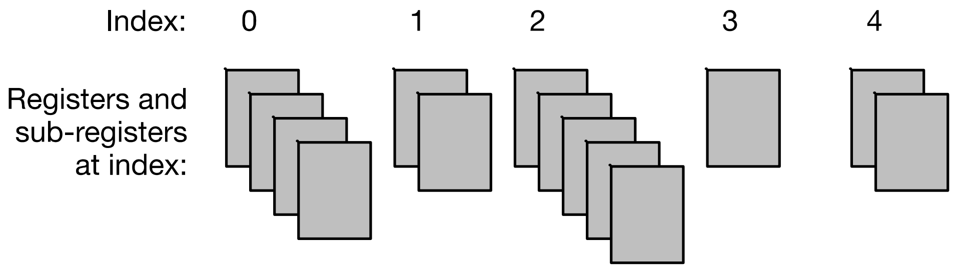

Figure 2 - Show remapped registers which create sub-register runs (clusters)

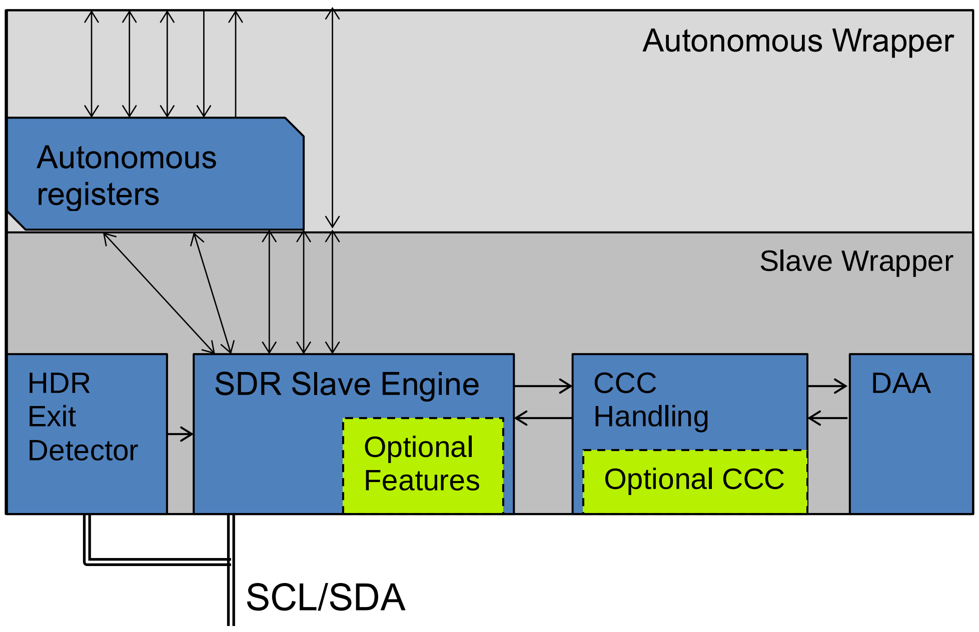

Figure 3 - Showing organization of autonomous block.

2. Autonomous Register access Slave interface

For devices with a programmable core, such as MCU or DSP, the APB interface would be used or the full interface (providing its own register interface). Likewise, for slaves which already have a way to handle register automation, they would use full or the mini wrapper.

The autonomous register access block works with the mini wrapper to provide support for a pure register scheme. A pure register scheme divides into 4 types of I3C bus accessible registers (aka commands):

- Read-only registers to the I3C bus (Master can read, but not write)
    - These are controlled by the device’s system logic, such as a sensor engine. That mechanism would normally also trigger an IBI (interrupt Master) on change, so that the Master could then read the newly collected data.
    - Other examples would be system state, and other registers of interest.
    - These would normally only be synchronized into the SCL clock domain if they can change while being read, unless the bits are atomic to each other. They do not have to be synchronized if safe-raw (e.g. changed when in STOP).
      The block does not synchronize. So, the system either needs to not change these when a transaction is in flight, or it has to synchronize as needed.

  - The mask can be applied, although it is usually simpler to simply form the net with 0s where no data.
  - These are selected with REG_WRITABLE[i]=0, REG_READABLE[i]=1, as detailed below.
- Read-Write registers (writable by Master, and Master can read back)
  - These are registers that the Master writes via the I3C bus, but also may read back. See also the 2-deep read/write model below as well as blended.
  - The system may interrogate these registers, although it should either use synchronization for safety or only read when not in a write transaction (which may be changing). When changed, the system is notified which either on a register-by-register basis or just start of runs using REG_RULES[3].
  - The Mask controls which bits are writable (and so readable).
  - These are selected with REG_WRITABLE[i]=1, REG_READABLE[i]=0, as detailed below.
- Blended read and write registers (Master writes some bits, and system controls some for read)
  - This means some bits are from the system and some from the bus. This allows for mixed schemes.
  - These are selected with REG_WRITABLE[i]=1, REG_READABLE[i]=1, and REG_BLENDED[i]=1, as detailed below.
- 2-deep read and write registers (Master writes a WO reg, system controls a RO reg with same index)
  - This model has the bus writing one set and reading a different set written by the system
  - This allows for things like write-clear of status as well as write request and poll to see if accepted. The system can copy bits over from WO to RO as needed.
  - These are selected with REG_WRITABLE[i]=1, REG_READABLE[i]=1, and REG_BLENDED[i]=0, as detailed below.

A register index which is neither writable nor readable is an empty hole - will not read nor write.

The model that the Master uses for Read:

S	addr/W	reg_index (aka CMD)	Sr	addr/R	Data return	Sr or P

The model the Master uses for Write is as shown below:

S	addr/W	reg_index (aka CMD)	Data from Master...	Sr or P

The register interface is setup to allow write and read, with special provisions for "runs". A run has different meaning for write vs. read:

- A read run means that if they perform a read, it will return as many registers are in the RUN, stopping at the 1st 0 (RUN[index]==0).
  - This may be as small as 1 byte register or may be dozens.
  - Note that if REG_RULES[0]==1, a Read will be NACKed on a read of a reg with RUN[index]=1.
- A write will normally go on as long as the Master pushes, ignoring the data if register is not writable.
  - Note that if REG_RULES[1]==1, a Write will be NACKed on a write of a reg with

RUN[index]=1, but only if the data is a separate address header. If after the reg/command, then it will simply be ignored (but error recorded).

- Note that if REG_RULES[2]==1, then write continuation will be ignored starting with the 1st RUN[index]=0 or a missing register.

2.1 The autonomous model and how it relates to modularity

As shown in Figure 4, the registers are bolted onto the Slave Engine wrapper. These registers are such that WO registers (from I3C Master) are in SCL clock domain, and RO (to I3C Master) are in the system clock domain. The only clock-crossing element then is the notification of change and the IBI request (if used).

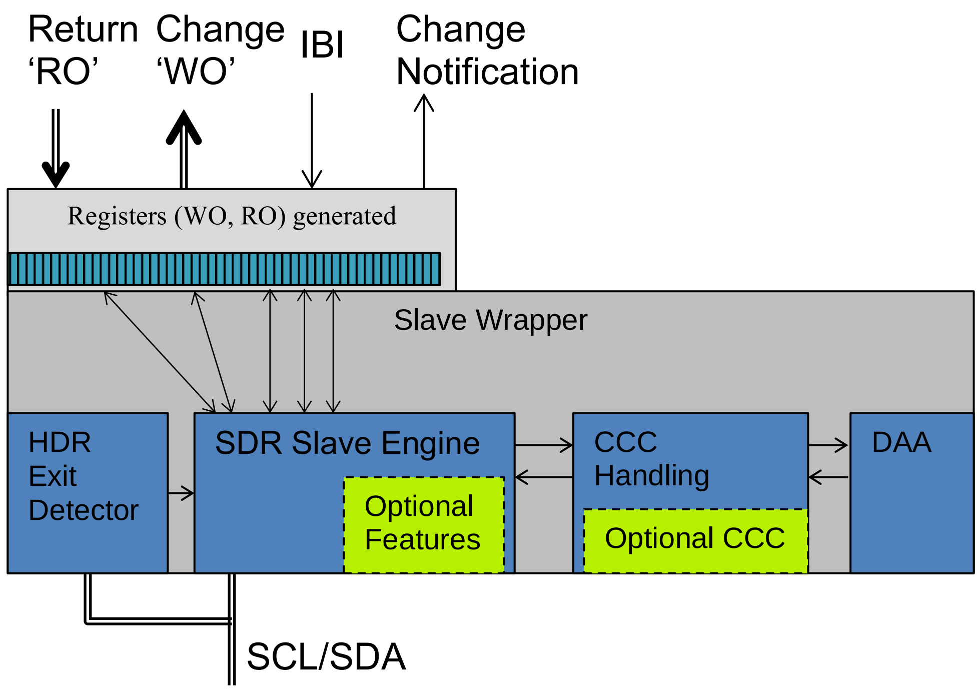

Figure 4 - Showing Autonomous registers - no CDC

2.2 The Magic Register(s) model – including unhandled CCCs if enabled

Along with the normal model of defining registers which are locally writable and locally or system readable, there is also a special model for active registers defined by the system. These are called Magic registers since the autonomous block has no insight into their operation.

These are only available if using the auton_full_wrap wrapper. They are tied off in the normal wrapper.

The autonomous block masters an 8-bit bus using the SCL and SCL_n clocks to allow system logic to handle these. They must be handled in SCL timing, so considerations of STA and S/H path timing should be factored in by this system logic – e.g. keep within the wrapping hierarchy and minimize load on the SCL clock.

The Magic register or registers are defined by masking indexes and then matching within that mask. This allows both a single register index as well as a range which is a power of 2, aligned to a power of 2. The magic register(s) may be rea/write or write-only, with write-only implying the normal readable register model would be used for read back – since read is not normally "active".

The bus signals are defined as follows such that all nets should be sampled on mr_clk_SCL rising, and read data should be available before mr_clk_SCL_n rising:

- `output        mr_clk_SCL;    // SCL clock we use (see also SCL_n)`

```
•  output [7:0] mr_idx;       // current register index (auto inc)
•  output [7:0] mr_next_idx;  // next index after trans_done
•  output       mr_dir;       // this is 0 if write, 1 if read (note timing)
•  output       mr_trans_w_done;// pulses for a cycle; at end of wdata (not idx)
•  output       mr_trans_r_done;// pulses for a cycle; at end of rdata
•  output       mr_trans_ddrcmd;// DDR CMD phase
•  output       mr_trans_ccc_done;// if enabled: pulses for each CCC cycle
•  output [7:0] mr_wdata;     // data with mr_dir=0 and trans_w_done=1
•  input  [7:0] mr_rdata;     // live data from system – change on trans_r_done
•  input        mr_rdata_none;// is 1 if no read data - will NACK if 1st
•  input        mr_rdata_end; // 1 on read if last byte
Note: below signals same as osync but not synchronized – in SCL domain
•  output       mr_bus_new;   // pulsed on match to us (note delay)
•  output       mr_bus_done;  // pulsed on START and repeated START
•  output       mr_clk_SCL_n; // inverted as valid clock (when not scan)
```

The mr_idx and mr_next_idx ports will be stable long before the mr_trans_w_done or mr_trans_r_done pulse; those pulses should only be sampled on SCL rising – they change on falling. Note that bits 7, etc, of mr_idx will be constant 0 if max reg is limited before 255.
If the Magic reg model is only for writes, with reads using the normal REG_READABLE approach, then the text below does not apply. Otherwise, the next part is how reads are handled:

> The mr_dir will also be stable, but for read, it will go to 1 after the Slave is already returning read data to the Master. So, it is critical to understand that the mr_trans_w_done and mr_trans_r_done signal occurs after the transaction (for both write and read). This means that the system must have read data ready before requested, or use mr_rdata_none to indicate not available (and will NACK the header, and also assert the raw_tb_reg_err signal). If used during data, it will work like _end but also signal the underrun error.
> The signal mr_rdata_end must be used for i3c reads to end the read – the master may terminate before, but the Slave is expected to end normally. This is to be released only on mr_bus_done.

For CCC, when enabled for unhandled, the mr_idx will contain the 8-bit CCC command code and mr_next_idx will have no meaning. This is only for mr_trans_ccc_done. The net mr_trans_ccc_done will pulse for each byte of data associated with the CCC, with mr_idx being the CCC byte itself. There will not be an mr_bus_new, only mr_bus_done with CCC. So, between them there will be mr_bus_done.

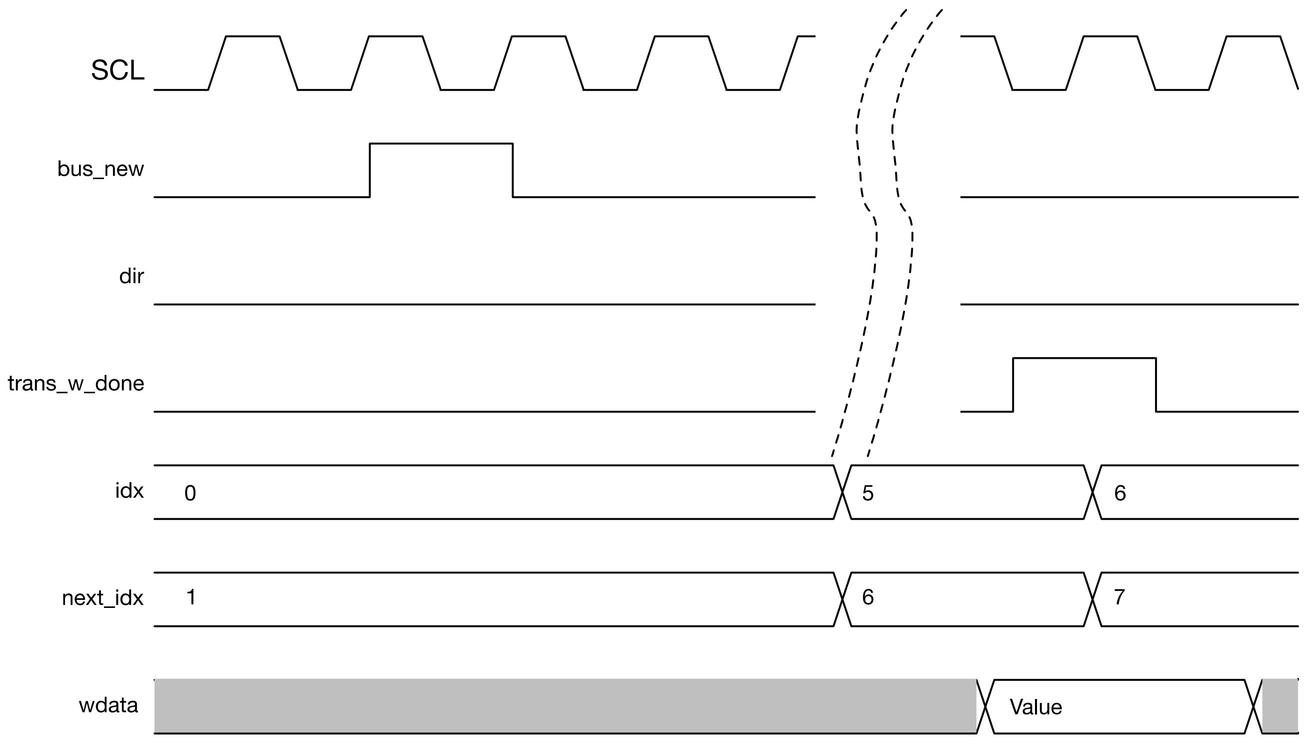

Figure 5 - Shows Write from Master through magic register bus

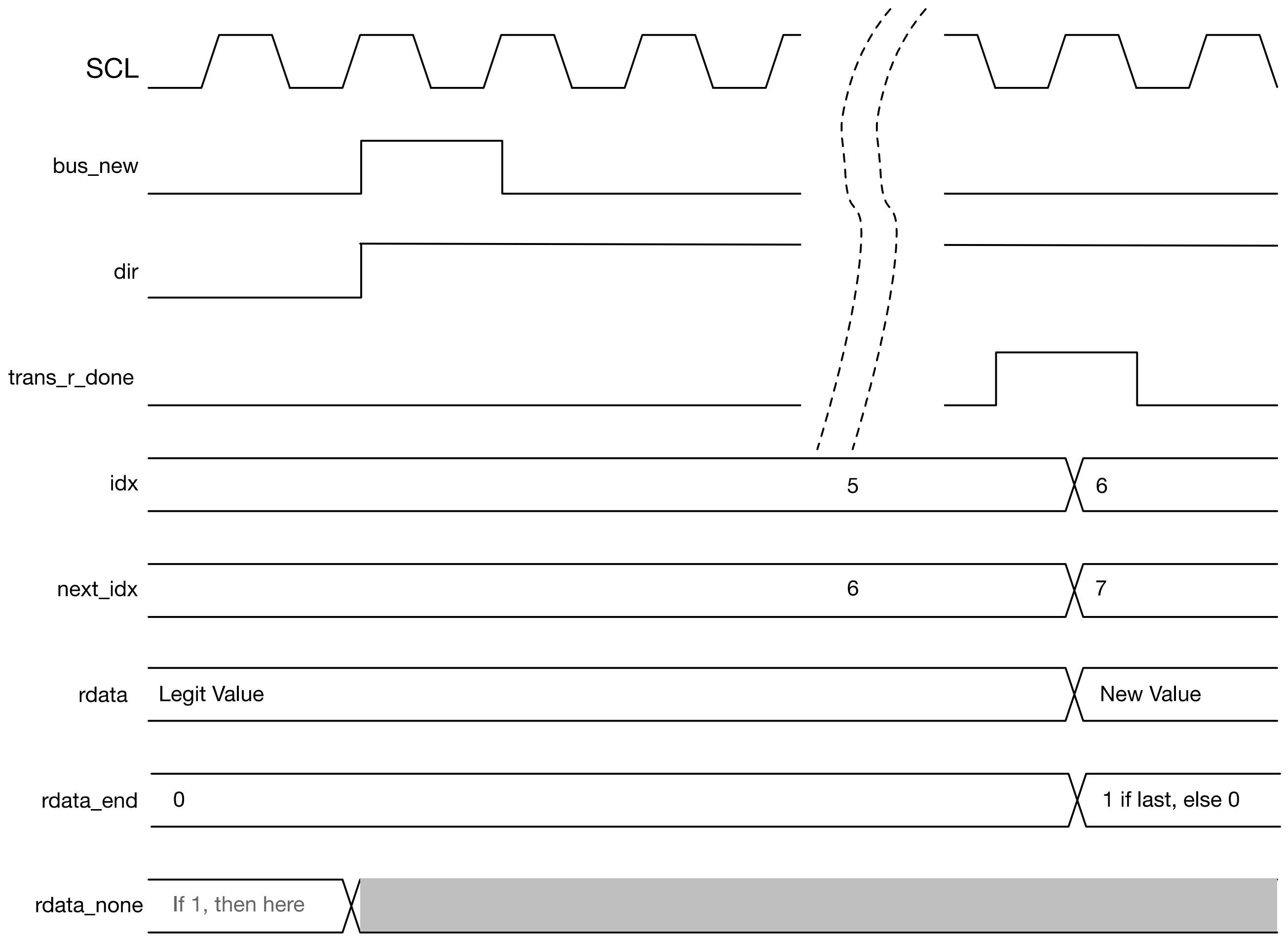

Figure 6 - Shows Read from Master through magic register bus

2.3 Parameters to define and control the block

The whole mechanism is handled via an RTL generate scheme, and so parameters control all rules as follows. Each bit group parameter listed is MAX_REG wide.

Parameter	**Meaning**	**Details**	**Affects**

REG_RULES	Sets certain access rules	[0]=1 if Write not allowed except starting when RUN[index]==0 [1]=1 if Write is ignored as soon as run hits RUN[index]=0 (to protect against over write) [2]=1 if only emit “touch” on RUN[index]==0 regs (to save flops). Would not catch mid-run starts [3]=1 if only emit “touch” on Write vs. on Write and Read. [4]=1 if reset index to 0 on S (not Sr), else read would continue from where left off. [5]=1 if using MAGIC register model [7:6]=DDR index approach: 0: Normal CMD=index (0 to 127) 1: CMD=index except if CMD=127, then next byte. 2: Index is in 1st byte. See also REG_DDRCMD_WRIDX.	All other params below.
MAX_REG [7:0]	Last register index	For example, if MAX_REG=63, then there are 64 register indexes from as [MAX_REG:0] or [63:0]. This is not how many registers there are since there may be holes. So, you could set to 127 and have 4 regs at 0-3 and then 4 more at 124-127, for a total of 8 regs over that index range.	All other params below.
REG_WRITABLE [MAX_REG:0]	Each bit is 0 if not writable, else is 1 if writeable.	A 0 implies read-only unless it is also not readable. If neither readable nor writable, it does not exist and the nets are not used and the flops are not created. A 1 means it can be written and can be read back if REG_READABLE is 0. If REG_READABLE is 1, then it is a 2 deep style.	Each register.
REG_READABLE [MAX_REG:0]	Each bit is 0 if not readable from system, else is 1 if readable.	If 1, it is provided by system. If REG_WRITABLE is also 1, then it is 2 deep. If 0 and REG_WRITABLE is 1, then it is read/write from the stored data of the writable reg. If neither readable nor writable, it does not exist and the nets are not used and the flops are not created.	Each register.

REG_RUN [MAX_REG:0]	Each bit is 0 if the start of a run (or not part of a run), else is 1 if allows continuation from previous.	A run means that a read will return as many bytes as are in the run. So, if registers 1,2,3,4, and 5 all have a 1, but 6 has a 0, then a read from 0 will return 0-5 or 6 bytes.	Each register.
REG_MASK [MAX_REG*8:0]	Each 8 bits is a mask. If a bit in the mask is 0, then no flop is created/used if writable and the net bit is not used and 0 is returned.	Allows allocating/using only as many bits as needed.	Each register bit.
REG_BLEND [MAX_REG:0]	Each bit is 0 if normal, else 1 if that register is blended.	To be blended, REG_WRITABLE and REG_READABLE are both set. The REG_MASK[i] applies to the writes from Master, and ~REG_MASK[i] applies to the read.	Each register
REG_REMAP [MAX_REG*8:0]	Each 8 bits is an index remap or 0xFF if no remapping.	Is normally only used to form clusters. So, can have a group of regs at 0, a group at 1, a group at 2, and do this by mapping down the 1st in the group from a higher number (e.g. 0x20, 0x24, 0x30).	Each register
REG_DDRCMD_WRIDX	Provides DDR CMD value to indicate index-only	This is used with REG_RULES[7:6]==2 such that it indicates which CMD indicates index-ony in next byte (so will be read following). This is because DDR sends a pair of bytes.	Just one

If REG_RULES[5]==1, then the following parameters are used as well for the Magic register(s).

Parameter	Meaning	Details	Affects
MAGIC_RULES	Controls use	[0] = 0 if magic used for read and write. 1 if not used for Read, so then make REG_READABLE for those so can be read back using normal model. [0] = 1 if unhandled CCCs to be passed through the magic regs bus. 0 if they are ignored.	Magic and overall use
MAGIC_MASK	Mask to select which indexes are covered.	0xFF means exact match, 0xF0 means 16 regs from mask.	Magic reg(s)
MAGIC_MATCH	Match index or indexes based on mask.	Match is (index&mask)==match. So, if mask is 0xF8 and match is 0x30, then magic regs are from 0x30 to 0x37.	Magic reg(s)

2.4 Organization of the block

The SDR Slave wrapper model is the same as with the I3C peripheral, and so that specification should be consulted for details.

The only added part is the Autonomous register mechanism and its wrapper.

3. Pin list for the Autonomous block to bolt into a System

The wrapper interface includes both required elements and then optional. Most optional elements are controlled by parameter (e.g. cf_ config by net vs. by parameter), but some optional ports are just information.

3.1 Clock crossing considerations

The autonomous block in general has no clock crossing and runs purely on the I3C (and i2c) SCL bus clock. It is mostly self-contained to provide a very easy to manage interface. There are 4 exceptions:

1. It provides a notification scheme using 4-phase handshakes between the SCL domain and the system domain. This allows the system to respond to events and know when it is safe to access data.
2. It provides an IBI request port to allow pushing an interrupt from the System domain.
3. It has wide nets crossing the domain, but these are protected against metastability using the other flagging mechanisms already described. That is, the registers are used “raw”, but change is protected against using 4-phase handshake flags to notify the system when the bus is actively changing things.
4. There is a bus available clock fed in to provide timing needed to wait before forcing an IBI.

What is notable is that these are all very lightweight CDC cases.

3.2 Required ports

The 1st set are RST and system clock (only used for a few synchronizers) followed by pin/pad control and input.

```
input               RSTn,             // reset from system
input               CLK,              // system clock for regs and sync
input               pin_SCL_in,       // SCL: normally clock in, data if ternary
input               pin_SDA_in,       // SDA: M->S
  // Note: next 3 are used special when PIN_MODEL==`PINM_EXT_REG
  //       They feed 3 flops placed close to the pads
output              pin_SDA_out,      // SDA: S->M on read
output              pin_SDA_oena,     // SDA: S->M on read
output              pin_SDA_oena_rise,// 0 if not ext_reg, else part oena D in
```

Now the autonomous register interface. We start with system notification in CLK domain of transaction started to/from us (this slave) as well as done (completed). We then have touch notification and data. The any_touch is synchronized to CLK to notify when transaction done and so touch can be sampled. The raw touch data is raw because it is only sampled when told it is safe (via any_touch and not new trans started). Each iraw_touch_OK bit is a reset to the corresponding touch bit, so is just pulsed 1.

Note that touch will be only start of runs if RULES select, else per reg.

```
output                  osync_started,    // pulse 1 when new trans started for us
output                  o_read,           // 1 if started trans is read, else write
output                  osync_done,       // pulse 1 if end of read or write (for us)
output                  osync_any_touch,  // pulse 1 if any reg R/W, when osync_done=1
output   [MAX_REG:0] oraw_reg_touch,      // bits=1 if reg been R/W (held until OK)
input    [MAX_REG:0] iraw_touch_OK,       // sys accept touch pulse – RST touch bit
output [(8*MAX_REG)+7:0] wo_regs,         // regs written by Master - raw
input  [(8*MAX_REG)+7:0] ro_regs,         // sys provided regs read by Master - raw
```

3.3 Optional ports under parameter control

The 1st set are related to IBI if enabled. This includes the special Bus Available clock and then Ibi request.

```
input                   CLK_SLOW,         // clock to use for IBI forced
output                  slow_gate,        // 1 if may gate CLK_SLOW
input                   i_ibi_event,      // placeholder: hold 0
input                   i_ibi_req,        // Hold 1 to request IBI, until done (or cancel)
output                  o_ibi_done,       // pulsed 1 CLK when IBI is done
output                  o_ibi_nacked,     // pulsed 1 CLK when IBI was NACKed by master
input             [7:0] i_ibi_byte,       // IBI byte if needed. Hold until done
```

Now configurations used when parameters indicate needed.

```
input               cf_SlvEna,        // Enable signal – may be strapped 1
input       [7:0]   cf_SlvSA,         // Static address for i2c if [0]==1 (and enabled
input       [3:0]   cf_IdInst,        // Instance of ID when selected (not if partno used)
input               cf_IdRand,        // random partno of ID if used
input      [31:0]   cf_Partno,        // partno od ID if used
input       [7:0]   cf_IdBcr,         // BCR of DAA if not const
input       [7:0]   cf_IdDcr,         // DCR of DAA if not const
output      [1:0]   raw_ActState,     // activity state from ENTASn if ena
output      [3:0]   raw_EvState,      // event mask from ENEC/DISEC if ena
output      [7:0]   raw_DynAddr,      // dynamic address we got from Master
output      [6:0]   raw_Request,      // bus req in process (see state_req below)
```

3.4 Optional ports that are purely optional

```
    // next one is only used if SDA & SCL pads have i2c 50ns Spike
    // filters and they can be turned on/off via net.
output                  i2c_spike_ok,     // Is 1 to allow i2c spike filter
```