I3C peripheral Slave µ-Arch spec

Overview of details for Slave RTL

Revision history

Revision	Date	Description	Author
1.3	10/23/19	Added MAP_DA_AUTO and MAP_DA_DAA.	PK
1.2d	01/27/19 05/16/19 07/12/19 07/29/19	RSACT_CONFIG MMR and custom. New MMR params in ID_AS_REGS: CCC MASK, HDRCMD, and VGPIO control (CCC based). Added i2c HS mode enable. tIDLE is 200us and not 1ms. This is valid for v1.0 too.	PK
1.1e	11/24/2017 3/13/18 10/5/18 10/30/18 12/18/18	Added section on layered local resets. Added ENA_MAPPED/MAP_CNT param. Added RSTACT and Slave Reset. Has DMA_TYPE=3 clarified. Cleanup DMA_TYPE. Added DMA section and Ext FIFO details.	PK
1.0hi	1/30/17 4/15/17 5/08/17 6/7/17 6/29/17 10/19/17	Should now be the official 1.0 spec to match the logic. Clarified params around CCC. Added port list for external pad control. Added Hot join timing param. Added time control params. Added DMA_TYPE and details on DMA in SEL_BUS_IF; restructured section to use headings. ID Vendor ID can be an MMR. Time control GETXTIME freq and acc can be an MMR. ERROR_HANDLING param added. Added BCR description and DCR URL Added ID_AS_REGS[5] for retention DA. PIN_MODEL was not described in terms of param. Also Sync TC is now allowed.	PK
0.9c	8/22/16	New params, better drawings, autonomous reg interface (draft), external reg pad control. Cleared up Internal FIFO details. More DDR related and added some slow-clock info.	PK
0.8C	5/8/16	Param changes, more details on pieces, CDC explanation, etc. Major update. Added more details on wrappers (full and APB). A diagram for metastability. Reg block.	PK
0.6	4/3/16	Parameter list added. Still need ports	PK
0.1	10/11/15	1st draft – requirements and arch	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
- Organized forms of multi-master:
  - Slave which can request Master to allow it to message another Slave, yet not needing to generate its own clock – called peer-to-peer
  - Secondary Masters which can use clean handoffs between each
- 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 capable of operating on i2c buses
  - Also support for bridging (to i2c, SPI, UART, etc)
- High data rate modes also optionally available
  - Only Master and the specific Slave has to support – other Slaves no how to ignore
  - Has an HDR-DDR form which is about 2x the data rate of SDR (so about 20Mbps)
  - Has an HDR-TSP (ternary symbols) which are up to 3x the data rate (so about 30Mbps)

The I3C peripheral supports the full feature set, but uses parameterization to allow reduction of the logic to what is needed.

2. Block organization and configuration

2.1 Organization of the block

The Slave I3C block is composed of a base I3C SDR engine, some base registers, and then a number of optional blocks which can be parameterized in. **Error! Reference source not found.** shows the block diagram of the peripheral in its full APB implementation. **Error! Reference source not found.** shows how the peripheral components are layered and wrapped, such that a device can pull in only the layer wanted. What is not shown are the full effects of parameterization, but that is explained in detail in section 2.2.

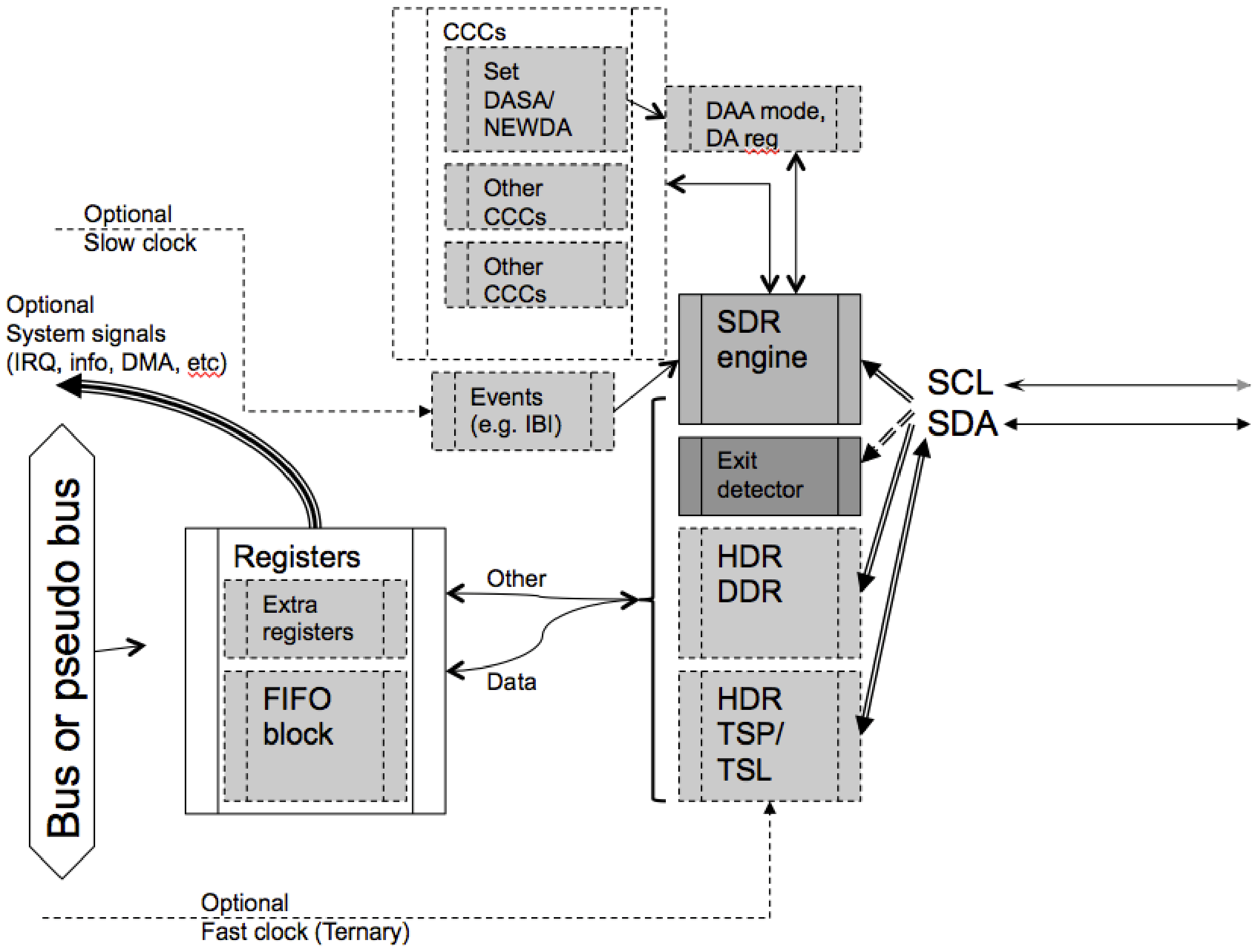


Figure 1 - Showing the block layout in terms of fixed and optional components

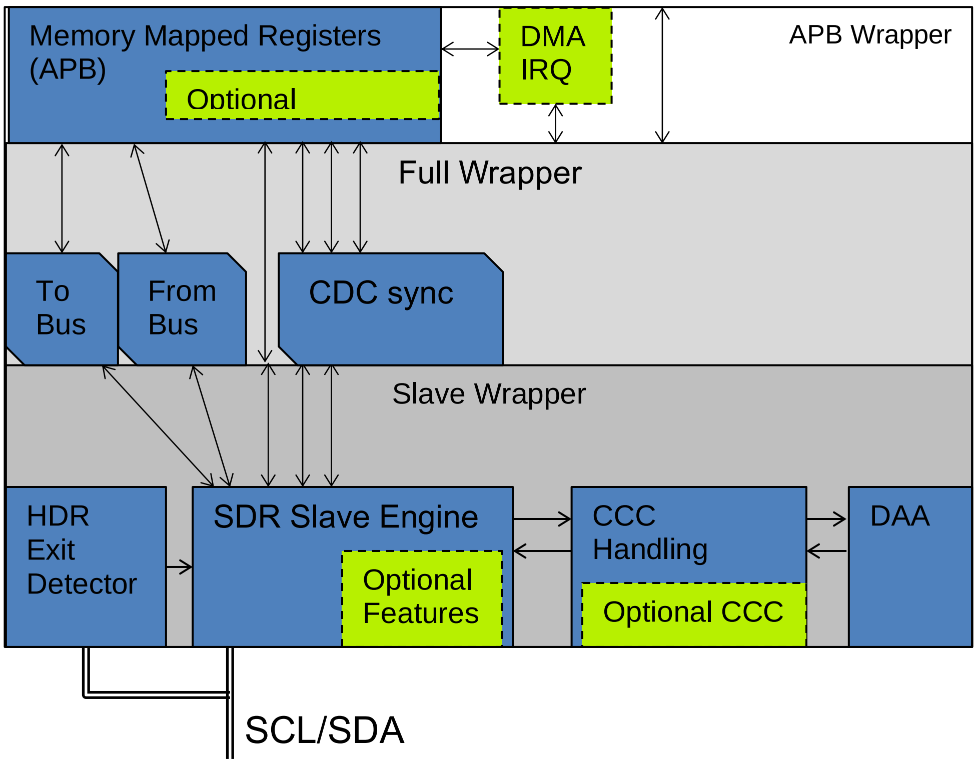


Figure 2 - Layers of wrappers to allow integration at different levels

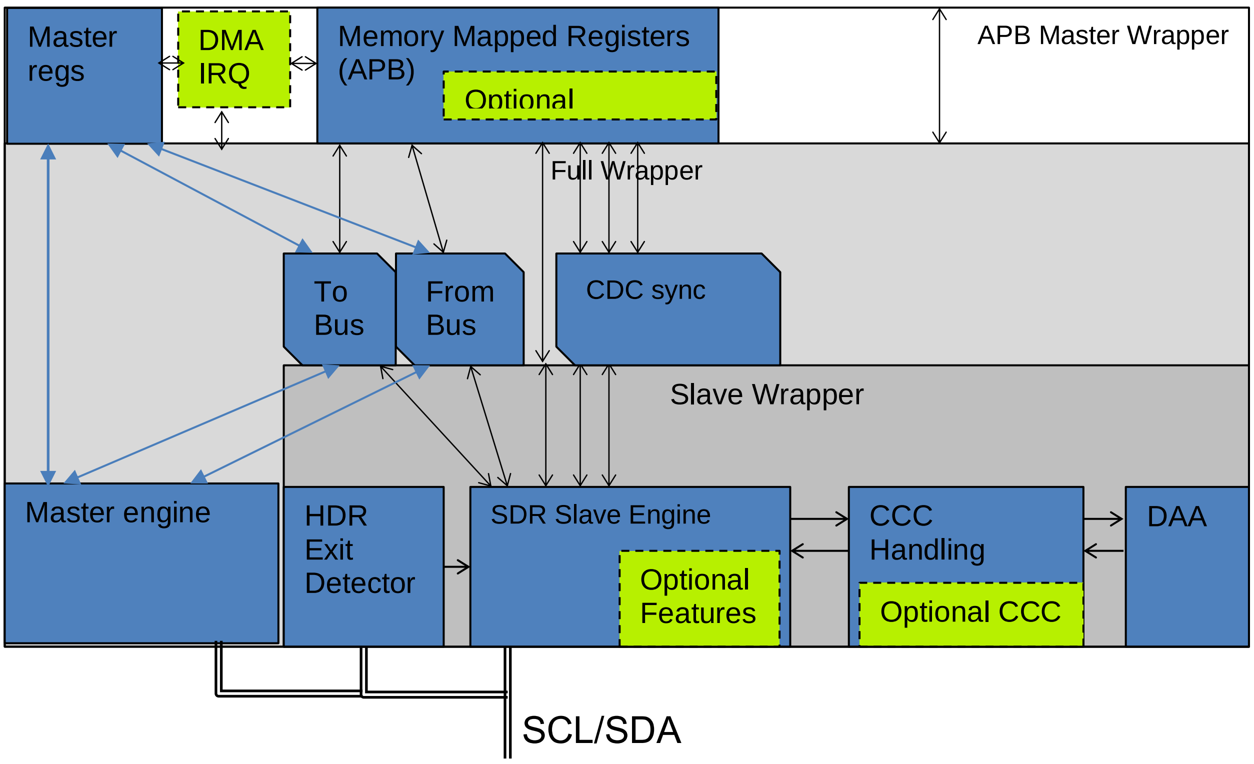


Figure 3. Showing Master+Slave wrapper

2.1.1 I3C Slave components

The model, built up from the inner-most out is as follows:

- I3c_sdr_slave_engine: Required. This is the core component of the peripheral.
  - It handles the common I2C/I3C protocol of frames wrapped in START and STOP, with possibly repeated START in the middle.
    - That is, it tracks START,header,data…,repeatedSTART_or_STOP
  - It handles standard I2C protocol, if the device has a static address.
    - So, it can match a header with that static address and process 9 bit data bytes (with ACK/NACK as 9th) in either direction. The data moves through the upper interface.
  - It handles the I3C SDR protocol for the devices assigned Dynamic address, once it is known
    - So, it can match a header with that Dynamic Address and process 9 bit data bytes (with Parity as 9th) in either direction. The data moves through the upper interface.
  - It handles the I3C SDR protocol for the broadcast 7'h7E in Write form.
    - So, it can detect a CCC, recognize if Broadcast or Direct, and recognize a certain set of known CCCs.
      - It then further recognizes when the CCC command "state" is done.
        For Broadcast CCC, that is next repeated START or STOP.
        For Direct CCC, that is next repeated START with 7E or STOP.
    - In particular, it recognizes the two "modal" commands:
      - ENTDAA. This is a required support and uses an upper layer block to handle the Dynamic Address assignment (see i3c_daa_slave). Further, it has to be tracked to end of the mode. Unlike normal Broadcast state, this Broadcasted mode allows repeated START,7E/R (read) and ends on repeated START,7E/W or STOP.
      - ENTHDRn. This set of matching CCCs enters into HDR mode. This then disables the SDR slave by "hold" and will not release until HDR Exit pattern is seen (and then STOP).
    - It also recognizes the other Dynamic Address related commands of Directed SETDASA and SETNEWDA, as well as the Broadcast or Directed RSTDAA.
      - These are not handled in the block directly, but are signaled out to the CCC handling block (see i3c_ccc_slave).
      - Note that SETDASA is special handled to change marked state based on seeing repeated START and the Static Address (if known) or the point-to-point address of 7'h01. The rest of the processing is done in the CCC handler.
  - It manages the SDA output and input, including open-drain output and push-pull. It does not touch SCL.
    - Depending on param, it either direct drives the pins (registered or combinatorial) or passes the combinatorial pre-control to a pad-control block close to the pads. See section 5.
- I3c_daa_slave: Required. Handles ENTDAA mode as well as owning the Dynamic address.
  - When i3c_sdr_slave indicates in DAA mode and i3c_sdr_slave indicates DAA active, this component runs a counter to emits its ID. DAA active is High if:
    - There is no Dynamic Address already defined (from SETDASA or this ENTDAA or a previous one).
    - The block has not lost the arbitration. That is, it is active from the repeated START, 7E/R (read) launching of the arbitration round. This component drives out the Open-Drain ID, and if not lost (see below), it stays active. If lost, it goes inactive until the next round.
    - This components emits the ID bits to the i3c_sdr_slave component while DAA active and the i3c_sdr_slave component pushes them Open-Drain. If the current bit is 1, so undriven, and the SDA is sampled as 0 on the rising edge of SCL, the arbitration is considered lost.
  - The counter counts out in 5 parts:
    - The 48 bit ID composed of VID and PID (part ID). This is emitted MSb 1st.
    - The BCR register emitted Msb 1st.
    - The DCR register emitted Msb 1st
    - If still DAA active, then this component has won. In that case, it reads the 7 bit Dynamic address from the Master followed by Parity.
    - If DA+Parity is OK, the address is ACKed in the following cycle. If it fails to get to an

        ACK, the dynamic address is not set and the component will try in the next round.
    - The ID and BCR and DCR may be from parameters (constant) or all or partly from nets or memory mapped registers.
    - The i3c_sdr_slave component will signal if RSTDAA is received as broadcast or directed to the Dynamic Address.
        - If so, the Dynamic Address is marked as invalid.
    - There is an interface to allow the CCC handler to modify the Dynamic address:
        - For SETNEWDA
        - For SETDASA using the static address (if any) or the point-to-point address of 7'h01.
- I3c_ccc_slave: Required, but only a small portion. Handles required CCC commands and may be configured to handle others – strongly suggested.
    - When i3c_sdr_slave indicates entered into a CCC command, this block decides whether to process it or ignore it (allowing the upper layer software to optionally process it).
        - SETNEWDA and SETDASA are always processed by this component. When matched, they push the Dynamic Address into the i3c_daa_slave component.
        - GETSTATUS is always processed by this component. It uses available knowledge to respond.
        - Parameters control what other CCCs are to be processed as explained below.
    - If basic CCC processing is enabled, then the block processes additional CCC commands:
        - GETID, GETBCR, and GETDCR are processed using the same data as was used by the i3c_daa_slave component.
        - ENEC and DISEC are processed to maintain event states.
        - SETASn is processed to set the activity state.
        - GETHDR is processed to return the HDR capabilities based on the parameters.
    - If MAX CCC processing is enabled, then the block process Max and Limit CCC commands:
        - SETMXWL and GETMXWL are used to control maximum write length. Usually a limit imposed by this device.
        - SETMXRL and GETMXRL are used to control maximum read length. Usually a limit imposed by the Master.
        - SETMXDS is used to report limitations related to data speed and read turnaround.
- I3c_exit_detector: Required. This is the HDR Exit Pattern detector, required whether HDR is supported or not.
    - This detects an HDR Exit Pattern and is used to release the SDR engine (i3c_sdr_engine) after an ENTHDRn CCC command has been processed.
        - That is, upon seeing ENTHDRn, the HDR mode is entered and the SDR engine is disabled in hold. The HDR Exit Pattern releases the hold.
- I3c_slave_wrapper: Required. This pulls all of the Slave components together per the parameters.
    - It also handles the clock domains related to SCL and SDR.
    - It also handles SDR hold related to HDR mode.
    - It also feeds up certain states and mode info to the upper layer.

2.1.2 I3C Top level Wrapper options

There are 3 possible top wrappers that can be used, depending on the requirements of the system. These match the parameters for this top level configuration. The details of these are in section 4.

1. APB Bus Wrapped with Memory Mapped Registers. See the I3C Peripheral Programmers model spec for details. This wrapper performs 4 tasks:
    a. It instantiates the memory mapped register interface, which uses APB for read and write by the APB master (e.g. processor core). Other bus types may be available.
    b. It instantiates the full_wrapper, which then instantiates the slave_wrapper, which is explained in the previous section.
    c. It handles the net vs. register mapping as controlled by parameters.
2. Full Wrapper (non-bus). This is the standard component when clock-domain-crossing and data handling support is wanted for system use (vs. autonomous – see next).

   a. It handles the data buffering/FIFO, depending on selections. This includes support for an external FIFO to be used with the buffer (ping pong) if needed.
   b. It instantiates the slave_wrapper, which is explained in the previous section.
   c. It handles CDC (clock domain crossing) for signals beyond data, including errors and “interrupts”.
3. Autonomous slave model. This is a thin layer over the i3c_slave_wrapper for uses reduced to the minimal I3C device. This is explained in section 5.2.
   a. Parameters are used to define a set of i3c “registers” as read-only, write-only/read-write to the i3c bus.
   b. The params also control run lengths and so on.
   c. The system side ports expose registers written by the Master as well as notification nets.
   d. The system side ports import nets to be readable by the Master.
   e. The system may trigger an IBI.
   f. The assumption is that a state machine is implemented in the system side to support these actions, although for very simple devices, it may be a direct map (e.g. current Ambient Light Sensor reading).
4. As noted in the previous section, the Slave Wrapper can be used for a minimal system that can run off SCL or that wishes to do its own clock crossing work.
   a. An example would be an light sensor which can use the SCL clock as the only clock and give a reading from polling alone.

2.1.3 I3C Master Components

TBD

2.2 Parameterization to control what the block does and does not do

The I3C Slave peripheral contains parameters to control what the block synthesizes to support.

In its absolute minimal form, the block only supports base I3C SDR, and operates byte at a time, relying on the system processor (software) to handle everything except the 4 most fundamental built-in (CCC) commands: ENTDAA, SETNEWDA, RSTDAA, SETDASA. That, is there is no FIFO, no extra CCC handling, no static address (i2c), no HDR modes, no IBI, etc. The system clock has to be fast enough to keep up with the data flow, since there is no clock stretching, although it can use the CCC to set max speeds.

Table 1 shows each parameter with a short explanation. Section 2.2.1 gives more details.

Table 1 - Parameters controlling Slave

Parameter	Related	Nets	Description
ENA_ID48B		Optionally: cf_ nets related to ID.	Automatic handling of Provisional ID for ENTDAA. Will also be used for corresponding GET CCC if enabled. Note that may use net or register PART_NO and Vendor ID
	ID_48B	-	Supplies all or part of 48 bit ID
	ID_BCR	-	Supplies Bus capabilities *register* when constant

	ID_DCR	-	Supplies Device-type *register* when constant
ID_AS_REGS		Will come from cf_ nets from the register bank.	Enables some ID nets and params to come from registers in the Application space. If not set, then the underlying parameter or net is used.
ENA_SADDR		Optionally ext_SlvSA or cf_SlvSA	Allows for i2c style static address to be used for SETDASA CCC and/or to work on an i2c bus.
	SADDR_P	-	Static address if fixed
ENA_MAPPED		-	Allows use of the MAPIDX/MAPSA fields of DYNAADDR and MSGMAPADDR register. Also, i2c 10bit address and other i2c features
	MAP_CNT	-	Number of SA/DA to allow after standard ones.
	MAP_I2CID	-	If not 0, is DeviceID for i2c use. The 3 bit revision comes from registers (ORed in, so optional)
	MAP_DA_AUTO	Control signals for DA_CHG	Enables Auto-MAP for auto DA from DASA, AASA, and/or DAA.
	MAP_DA_DAA	Map_regs if MMR	If DAA and not MMRs, then details per MAP slot.
ENA_CCC_HANDLING		Optionally raw_ and state_ data.	Allows block to support some CCC commands vs. system doing so. This should be set to at least BASIC if IBI used, unless the system will handle.
	RSTACT_CONFIG	SLVR related	Enables and controls RSTACT and SlaveReset support.
	MAX_RDLEN	cf_MaxRd	If enabled, handles max read length
	MAX_WRLEN	cf_MaxWr	If enabled, handles max write length
	MAX_DS_WR	-	If enabled, sets max data speed for write

	MAX_DS_RD	-	If enabled, sets max data speed for read
	MAX_DS_RDTURN	-	If enabled, sets the max read turnaround time
ENA_IBI_MR_HJ		Optionally reg_EvPend and reg_EvIbiByte, with outp_EvDet back	Enables IBI, Master Request, and Hot join events. Also allows for data bearing IBI. Works with registers to support.
	CLK_SLOW_BITS	CLK_SLOW	Indicates how many bits are needed for counter to get to Bus Available period. CLK_SLOW may be same as CLK or slower one.
	CLK_SLOW_MATCH	CLK_SLOW	Value to match for Bus Available. See also BAMATCH reg.
	CLK_SLOW_HJMUL	CLK_SLOW	Multiplier of Bus Available periods for Hot Join. Normally would be 10’d1000 to get from 1us to 1ms. NOTE: HJ is now 200us, but we still have 1ms reference for now. It only has to be good enough to ensure 2x100us is accurate enough for 200us or more.
	ERROR_HANDLING	CLK_SLOW	Allows for error checks: Read-Abort, and Reg write errors. The S0/S1 end mechanism is always on if CLK_SLOW available.
	ENA_TIMEC	CLK_SLOW_TC	Enables 1 or more time-control modes. Currently only Mode 0 is supported, which is bit 1. Also used to enable MMR for the freq and acc bytes of GETXTIME.
	TIME_FREQ_ACC	CLK_SLOW_TC	Defines 2 bytes with frequency*2 (to allow .5MHz resolution) and then accuracy in 1/10ths of a percent. So, 24 is 12.0MHz and 10 is 1.0%. If MMR to be used, this sets the reset value.

FIFO_TYPE		-	Controls details of FIFO if any, else 0 if none
	EXT_FIFO	ixf_* and oxf_*	External FIFO interface to allow for a FIFO block outside of the peripheral. This defines if one and if so, which of 3 supported interfaces.
	ENA_TOBUS_FIFO	-	Enables FIFO for sending data to Master from Slave. Sets depth as power of 2.
	ENA_FROMBUS_FIFO	-	Enables FIFO for receiving data from Master. Sets depth as power of 2.
ENA_HDR		-	Enables support for 1, 2, or 3 HDR modes (TSP and TSL are almost the same).
	CLK_FAST_SPEED	CLK	Speed of fast clock when needed for hdr ternary. Normally a multiple of 11 to 12.5MHz (e.g. 25MHz, 33MHz, etc).
SEL_BUS_IF		Bus, interrupt, DMA, and general info nets. Also, ixf_ and oxf_ if external FIFO is selected.	Enables what kind of “bus” interface this block has to the system, including what is supported.
	DMA_TYPE	dma_xxx	Indicates of Trigger or request style DMA. Default is request.
PIN_MODEL		pin_SDA_xxx	Normally combinatorial for SDA for lowest area/power. But, can be registered or external registered if timing is a problem.
BLOCK_ID		-	If not 0, provides a read only register at offset FFC.

2.2.1 Details of parameters

Parameters allow integration of other functionality as follows (order in terms of likely selection):

2.2.1.1 ENA_ID48B, ID_48B, ID_BCR, ID_DCR, ID_AS_REGS

Indicates how the 48-bit address is provided by the block or system code in response to ENTDAA CCC as well as GETPID. Allows degrees of freedom from purely hard-wired to from registers. The “Vendor Fixed value” is referred to as the PARTNO in this spec.

- ENA_ID48B is:

    - 1 if the ID_48B parameter contains the full 48-bit ID (obviously not Random PARTNO)
    - 2 if the ID_48B parameter contains the 48-bit ID except bits 15:12, which come from input net: [15:12] cf_IdInst or register – see ID_AS_REGS.IDINST.
    - 3 if the ID_48B only has the 15 bit MIPI Manufacturer ID, and the cf_Partno net or memory mapped register PARTNO has the 32 bit remaining component of the ID. This also allows for Random PARTNO via ID_AS_REGS.IDRAND.
    - 4 if the whole ID part comes from Registers or Nets using the cf_IdVid, cf_IdRand, and cf_Partno nets. NOTE: must set ID_AS_REGS[4] with 1 for Vendor ID from regs. Else, it will be 0.
    - 0 is disallowed and will cause a synthesis error
- ID_BCR and ID_DCR: contain the details of the bus and device config regs used in ENTDAA and from specific CCC requests unless ID_AS_REGS.IDDCR and/or ID_AS_REGS.IDBCR are set, in which case they come from the IDEXT register (cf_IdBcr and cf_IdDcr nets).
    - If BCR[0] is set, then ENA_CCC_HANDLING[1] should be set.
    - BCR[5] may be 0 and then the DDROK CONFIG bit can be used to enable or disable, as long as ENA_HDR[DDR] is also 1.
    - BCR bit meanings (same as i3c spec):
        - [7:6] = 0 if Slave only, 1 if Master+Slave, 2 and 3 reserved
        - [5] = 1 if HDR capable (DDR or TSP or both), 0 if not
        - [4] = 1 if Bridge device
        - [3] = 1 if Offline capable; means if in lowest power state, no response but retains the DA.
        - [2] = 1 if IBI with payload. May only be 1 if [1]=1 as well
        - [1] = 1 if IBI (interrupt)
        - [0] = 1 if has limitations so that GETMAXDS should be used by master to find details
    - DCR is defined: https://mipi.org/MIPI_I3C_device_characteristics_register
- ID_AS_REGS: used to force some values into application controlled registers/nets vs. parameters. If using a register wrapper such as i3c_apb_wrapper for a general purpose MCU, these nets will usually be registers (vs. param constants being used for them). Set as bits:
    - Bit [0]=1: IDINST is used to make the instance a register vs. a net. Only used when ENA_ID48B=2, so partno is from param but with instance separately. In both cases, cf_IdInst contains the value.
    - Bit [1]=1: IDRAND is used to allow the application to define the PARTNO as a random value vs. normal partno. Can only be used when ENA_ID48B=3 such that PARTNO is used. This is fed by cf_IdRand. The partno is cf_Partno.
    - Bit [2]=1: IDDCR is used to provide the DCR via the IDEXT register.
    - Bit [3]=1: IDBCR is used to provide the BCR via the IDEXT register. Not also implication of DDROK in CONFIG register for HDR-DDR control.
    - Bit [4]=1: Vendor ID (Manufacturer) comes from VENDORID register. This will only be used if the ENA_ID48B is 4 (ID_CONST_NONE).
    - Bit [5]=1: Allows DYNDDR register to be writable for retention recovery.
    - Bit [6]=1: Slave is enabled from release of reset. This should not be used if config settings are to come from MMRs unless it is known that the bus will be inactive for a while.
    - Bit [7]=1: Allow HDRCMD reg. This is still enabled by the CONFIG reg.
    - Bit [8]=1: Allow VGPIO reg; is used with map vgpio to control how it works.

- Bit [9]=1: Allow CCC_MASK which is used to mask on/off unhandled CCC groups
- Bit [10]=1: Allow ERRWARNMASK reg to mask ERRWARN bits that feed to STATUS.ERRWARN status bit.

2.2.1.2 ENA_SADDR and SADDR_P

Indicates if an i2c style static address or not how provided if so.

- ENA_SADDR is:
  - 0 if no static address (default min – only gets dynamic address)
  - 1 if static address by parameter (SADDR_P)
  - 2 if static address by net (ext_SlvSA).
  - 3 if static address by memory mapped register CONFIG (SW writes using cf_SlvSA)
- SADDR_P: used to provide a static address as a parameter if ENA_SADDR=1.

Note: if net is used (cf_ or ext_), bits [7:1] are the address and [0]=1 if the address is valid.

If static address is provided, it serves two purposes:

1. It allows the I3C Master to use SETDASA CCC to set the dynamic address from the static.
2. It allows the Slave to be used on an i2c bus.

2.2.1.3 ENA_MAPPED

If enabled, will allow for multiple Static address (if ENA_SADDR) and/or Dynamic addresses (always) to match for messages. This will enable the MAPIDX/MAPSA fields of the DYNADDR register. This also creates the MSGMAPADDR register. This is also used for most advanced i2c features except device-id, which is not normally used by itself.

For static addresses, the model is that the 1st static address is handled in the normal way (ie. ENA_SADDR). Additional ones can be added using the DYNADDR register with the MAPIDX=N (where N is not 0) and MAPSA=1.

For Dynamic addresses, the model is that the Master sets the 1st DA in the normal way. The SW may use one of two methods (or a combination):

1. Use the Bridge Targets CCC to set more DAs using the DYNADDR register with MAPIDX=N and MAPSA=0. This can set each address to match.
2. Copy the base DA set by the Master using the DYNADDR register with MAPIDX=N and MAPSA=0, and invalidate the base DA using the DAVALID=0 method (must be enabled). This can allow it to represent multiple instances of the same device or different devices using the memory mapped registers IDEXT or PARTNO. This may only be used if IDREGS[DAWR]=1.

The extra DAs or SAs can be disabled by using the same MAPIDX but with valid set to 0.

- ENA_MAPPED is:
  - Bit [0]=1: Allows multiple mapped addresses, else just 1
  - Bit [1]=1: for allowing i2c 10-bit address in map index 1
  - Bit [2]=1: for other extended i2c features like SW Reset (DeviceID from MAP_I2CID)
  - Bit [3]=1: for i2c HS mode detect. When detected, it sets a port to 1, which clears on STOP.
  - Bit [4]=1: for VGPIO mechanism – private, do not use.
    Note: reserves index 1, and cannot be mixed with bit[1]=1.
    Note: see also ID_AS_REGS for the VGPIO MMR enable to allow CCC based VGPIO.
- MAP_CNT is set to the number of mappable SAs/DAs to support (beyond the 1st built-in ones).
  - Note that 1 is reserved for 10-bit SA if enabled

- MAP_I2CID is set to the i2c Device ID 24 bit value if to be supported, else is 0.
  - Note that the lower 3 bits come from an MMR which is ORed into the bottom 3. So, is optional.
- MAP_DA_AUTO is used to control Automatic DA assignments from MAP slots. It can only be used if Map is enabled. Bits control
  - Bit[0]=DASA enable (for SETDASA)
  - Bit[1]=AASA enable (for SETAASA)
  - Bit[2]=DAA enable (for ENTDAA). If this is set, then remaining bits apply
    - Bit[6]=MMR which means DAA from registers vs. from MAP_DA_DAA
    - Bit[7]=DCR is to be replaced (all 8 bits) whether MMR or constant
    - Bit[12:8]=0 if PID is not replaced, else number of bits to replace.
      NOTE: It is highly recommended that this be a power of 2 (1, 2, 4, 8). It may be up to 10 if DCR replaced, else up to 18.
- MAP_DA_DAA is a string of bits for DAA if not MMR. So if Bit[2]=1 and Bit[6]=0. The format for each slot (MAP_CNT) is:
  - Bit[0]=1 if enabled, else 0 if not enabled for this slot. Controls if participates.
  - Bit[8:1]=DCR if enabled
  - Bit[n:9] or [n:1] depending on if DCR for PID. So, PID replacement of PID[m:0] based on MAP_DA_AUTO length. Normally would be 4 or 8 bits.

2.2.1.4 ENA_CCC_HANDLING

Indicates which (if any) CCC to handle in the block.

Note that this is exclusive of ones always supported such as ENTDAA, SETDASA, SETNEWDS, RSTDAA, GETID, GETBCR, GETDCR, GETSTATUS, GETHDR (cap), and ENTHDR (needed whether HDR supported or not, so we know entering).

- ENA_CCC_HANDLING is a set of bits such that all bits 0 means no CCC commands are supported by the block (other than handled due to other params). The system must handle them itself when not optional, and may handle them if optional. Otherwise the bits are:
  - Bit [0]=1: Block handles Events, Activity states.
    - Note: the events and other details are exposed via the output nets raw_ActState and raw_EvState. They will also show up in the STATUS register (if used).
    - This should be set if IBI used since ENEC/DISEC required if so.
  - Bit [1]=1: Handles Max Rd and Wr length and max SCL (data speed) if needed.
    - Note: if enabled, the read and write length are on the output nets cf_MaxRd and cf_MaxWr as well as in the register MAXLIMIT (if a register). The results from the master will be in the outpflg_MaxRd and outpflg_MaxWr. The default max the Slave wants is defined by MAX_RDLEN and MAX_WRLEN params and those initialize the registers. Do not enable this unless the cf_ nets are provided for.
    - Note: the details of max data speed are in MAX_DS_xxx param (see below).
    - This should be set if the BCR bit 0 is set (limitations).
  - Bit [2]=1: Support GETSTATUS fields like PendINT via the CRTL register's PENDINT and ACTSTATE fields.
  - Bit [3]=1: Support GETSTATUS VendorReserved field via the CRTL register's VENDINFO field.

2.2.1.5 RSTACT_CONFIG

This controls the RSTACT CCC support as well as the signaling related to Slave Reset.

- The RSTACT is 20 bits:
  - Bit [0]=1: Enables support for Slave Reset
  - Bit [1]=1: Enables MMR support for time to recover (vs. this param)
  - Bit [2]=1: Allows a custom RSTACT value (0x40 to 0x7F). Bits [25:20] encode the value to OR into 0x40.
  - Bits [4:11]: The time needed to recover from peripheral reset in ms. 0 is default.
  - Bits [12:19]: The time needed to recover from a system reset in seconds.
  - Bits [25:20]: The custom RSTACT value (added to 0x40) if bit [2]=1

2.2.1.6 MAX_RDLEN, MAX_WRLEN

These hold the max lengths to initialize the registers to when ENA_CCC_HANDLING[1]=1. The max is 12 bits each.

- MAX_DS_WR, MAX_DS_RD, MAX_DS_RDTURN: these are only used if ENA_CCC_HANDLING bit [1]=1.
  - MAX_DS_WR contains the byte to return as the 1st byte of the GETMAXDS CCC.
  - MAX_DS_RD contains the byte to return as the 2nd byte of the GETMAXDS CCC.
  - MAX_DS_RDTURN contains 0 if none, else the 24 bit value for the 3rd, 4th, and 5th bytes.

2.2.1.7 ENA_IBI_MR_HJ, CLK_SLOW_BITS, CLK_SLOW_MATCH, ENA_TIMEC, TIMEC_FREQ_ACC

The ENA parameter is non-0 when the Slave will use the IBI, MR, or Hot Join mechanism. The parameters are interpreted as:

- ENA_IBI_MR_HJ:
  - Bit [0]=1: the Slave will use IBI and pend that via the CTRL register or uses reg_EvPend (see REG_IBIREQ). Note if 1, then should set ENA_CCC_HANDLING[0] as well.
  - Bit [1]=1: if the Slave uses IBI (bit [0]=1), it will be data bearing (a byte follows). This allocates the IBIDATA field of the CTRL register. It will use reg_EvIbiByte.
  - Bit [2]=1: the Slave will use the Master Request (P2P or 2ndary master) and pend that via the CTRL register. Again, using reg_EvPend.
  - Bit [3]=1: the Slave will use the Hot Join request and pend that via the CTRL register or uses the opti_hj_request (see REG_HJREQ). Again, using reg_EvPend.
  - Note: if any of these are set, the input net CLK_SLOW must be connected using a fixed invariant slow clock or it may be PCLK/CLK. See also the CLK_SLOW_x and slow_gate (gates slow clock when not needed).
  - Bit [4]=1 means BAMATCH register is used vs. CLK_SLOW_MATCH to set the 1us Bus Available counter match on the slow clock. This is only needed when the Slow clock is variant (e.g. PCLK), else CLK_SLOW_MATCH is sufficient.
  - Note: the results are carried in outp_EvDet.
  - Bit[5]=1 means Slave EXTDATA for IBI is in a separate 2-entry FIFO
  - Bit [6]=1 means Master read of IBI data goes into a separate FIFO (size can be set)
- CLK_SLOW_BITS is used to allocate bits for a counter to count out the 1us of Bus Available condition. It is only used if the full wrapper at least is used since it counts from the slow clock. Note that the Slow clock

may be the main clock or a fixed invariant slower clock.

- CLK_SLOW_MATCH is used to match the count within CLK_SLOW_BITS for the 1us (or more) of Bus Available condition. If the Slow clock is a variant clock (e.g. PCLK and can be changed dynamically), then the register should be used (BAMATCH), and this only forms the reset value for the register. Note that the value is 0 relative. So, 25MHz would be 24 to get 1us (40ns per clock, 1000/40=25, but is 0 rel, so 24).
- CLK_SLOW_HJMUL is used if Hot-join is enabled. This indicates the number of Bus Available periods (regardless of how matched) to get to 1ms (the old tIDLE). Since tIDLE for Hot Join is now 200us, this is just used to ensure the 100us and 60us counts are >= 100us and 60us. The default is 1000, which is 1000x1us=1ms. Note that this is factored to form 10x 100us counts and it should be rounded up to be at least 1ms (OK to be more, not OK to be less).
  NOTE: HJ timing is 200us in v1.1 (and is valid for v1.0 as well), so only use this value to be sure 200us or more (but using 1ms ref).
- ERROR_HANDLING bits indicate what error handling to support.
  - Bit [0]=1 if Read-Abort happens if read stalled more than 100us. This is on by default if IBI is enabled.
- ENA_TIMEC indicates which time control modes are supported as bits:
  - Bit [0]=1 if Synchronous is supported; NOTE: supported as an external block. This enables the port to to that block.
  - Bit [1]=1 if Asynchronous Mode 0 is supported.
  - Bit [2]=1 if Asynchronous Mode 1 is supported; NOTE: not supported at present
  - Bit [3]=0 reserved
  - Bit [4]=0 reserved
  - Bit [5]=1 if TCCLK register should be created. This will allow dynamic control of the GETXTIME freq and accuracy values vs. constant. Note that the constant param normally should be set to set register reset value.

- TIMEC_FREQ_ACC provides 2 bytes as upper and lower and relates to CLK_SLOW_TC:
  - Upper Bits [15:8] = Frequency in 0.5MHz units. So, 12MHz is 24 and 12.5MHz is 25.
  - Lower Bits [7:0] = Accuracy in 0.1%. So, 1.3% accurate is 13.

2.2.1.8 FIFO_TYPE, EXT_FIFO, ENA_TOBUS_FIFO, ENA_FROMBUS_FIFO

These are sed when FIFOs are supported for internal or external use. Note that a minimal ping-pong FIFO is always included when full wrapper is used.

- FIFO_TYPE: this selects which kind of FIFO to use and other FIFO related controls
  - [1:0] = 00 if no FIFO, 01 if internal FIFO, 10 if external FIFO
  - [2] = Add holding buffer to to-bus to avoid direct muxes when internal FIFO used. This is not used when no FIFO or external FIFO – not needed.
  - [3] = Add holding buffer to from-bus when **no** internal FIFO, which adds 1 more buffer's worth. If used with external FIFO, separates mux from external FIFO path.
- EXT_FIFO: this controls use of an external FIFO block to support data buffering past 2 bytes and when not using the internal FIFO. This must be 0 if FIFO_TYPE[1:0] != b10.
  This indicates which type of external FIFO is to be used. In all cases, the data is moved by byte, but the actual FIFO can roll these up to half-words or words if wanted.
  - 0 = none

  - 1 = normal available/free tracking.
  - 2 = request model internal
- ENA_TOBUS_FIFO is set to number of elements as a power of 2, or 0 if not used; this is the data to send to the Master from this Slave. This must be 0 if FIFO_TYPE[1:0] != b01
  - This may be any power of 2 greater than 1, so 2=4 bytes, 3=8 bytes, etc.
  - If more than 16 or so are needed, it is recommended to use a separate FIFO scheme which uses a more dense memory than just flops
  - This is also used for CCC commands that are not handled by the block, and may also be used for ID arbitration when not handled by the block.
  - FIFO_TYPE[2] should be set to 1 to limit muxing into the engine
- ENA_FROMBUS_FIFO is set to number of elements as a power of 2, or 0 if not used; this is the data to get from the Master to this Slave. This must be 0 if FIFO_TYPE[1:0] != b01
  - This may be any power of 2 greater than 1, so 2=4 bytes, 3=8 bytes, etc.
  - If more than 16 or so are needed, it is recommended to use a separate FIFO scheme which uses a more dense memory than just flops
  - This is also used for CCC commands that are not handled by the block.

2.2.1.9 ENA_HDR

Enable bits for each HDR to be supported. Note: HDR-TSP/TSL removed due to IPR issues.

- Bit [0]=DDR to be supported. Will indicate when active via o_hdr_act_ddr and via STATUS register.
- Bit [1]=BT to be supported with v1.1.
- Bit [2]=reserved.

2.2.1.10 SEL_BUS_IF and DMA_TYPE

Select bus allows controlling the “bus” model as bits. If the param is 0, then there is no bus interface and no memory mapped registers. By default, bits 0 and 1 are always set when using the APB wrapper level. The others are dependent on what is needed.

- Bit [0]=1: APB type bus: PSEL with address and PENA|PSEL with data in 2nd cycle. This is used to read and write registers (as defined below).
- Bit [1]=1: expose all general output types as registers vs. as nets only; this includes things like dynamic address, state, activity level, etc.
- Bit [2]=1: Support Interrupt model with INTENSET/INTENCLR/INTSTAT
- Bit [3]=1: Support DMA requesting model. See also DMA_TYPE. The DMA model works in one of 3 ways:
  - Trigger, with DMA_TYPE[0]=1. This means that when the TX or RX trigger occurs (based on DATACTL trigger level). The corresponding DMA request is pulsed for one PCLK and then not re-armed until the trigger goes away.
  - Request style which sets request High on the corresponding trigger (which may be empty/not-full), and then holds level High until the corresponding TX FIFO is full or the RX FIFO is empty. There are two forms:
    - ACK is a 1 cycle pulse (normally) preceding the access of the WDATAB/H or RDATAB/H MMR. If the ACK is seen and the last value (RX last entry, TX 1 entry from full), request will de-assert. Else, it holds high.
    - ACK is a 1 cycle pulse (normally) with or following the access of the data MMR. In this case, the request holds high until RXFIFO empty or TXFIFO full.
      Note: ACK may be strapped 0 in this case.

  - 4-phase handshake with request and ACK, such that ACK is with/after the MMR access. The 4 phase works by Request high, ACK high, then Request Low, then ACK low. Then, Request may go high again if more data in RXFIFO or more space in TXFIFO.
- Bit [4]=1: Allow Halfword read and write data since quicker for processor and DMA

BLOCK_ID: should be defined for ID/revision register at offset 0xFFC (standard ARM ID).

DMA_TYPE is used to select if DMA trigger mode vs. request, as explained above under bit[3].

- 0= If Request/ACK 1 cycle before the access of the MMR.
- 1= unused.
- 2= If Request/ACK 1 cycle with/after the access of the MMR. The ACK may be strapped 0.
  Note: no "done" model anymore.
- 3= request/ack with 4-phase handshake:
  - 1. dma_req_xx is set to 1 by the IP - to indicate it wants the DMA to perform a transfer.
  - 2. dma_req_xx_ack is set to 1 by the DMA - to accept.
  - 3. dma_req_xx is set back to 0 by the IP in response to ack=1.
  - 4. dma_req_xx_ack is set back to 0 by the DMA - in response to req=0.
  - 5. Go back to 1 if more to do.

Note: the signal dma_req_tb_ack_last may be used to cause the last byte to be marked as END for to-bus. It should be strapped 0 if not used, else may be 1 with the write to WDATAB or WDATAH when it knows its count is 1 entry from done.

2.2.1.11 PIN_MODEL

The PIN_MODEL param is used to select whether the SDA is combinatorically fed from a cloud (for most area efficiency), from 2 registers (enable and output), or via a pad control block which is external to the peripheral and placed near(er) the pad. See section 5 for more details).
It is set to one of:

- 0=PINM_COMBO uses combinatorial feed for the output pin drive (SDA) from Slave.
  This gives smallest area and generally lowest power (avoiding fan in). Most modern process can do well for the 12.5MHz max, but if you are finding your SCL-in to SDA-out is worse than 12ns (exclusive of output SDA PMOS/NMOS, so from Schmitt input on SCL to gate to driver on SDA), then you should use one of the registered ones.
- 1=PINM_REG uses registered controls for SDA (out-enable and out) with only 3 gates of logic after the Qs (for emergency cases).
  This registers 1/2 clock early and so can give better timing to the pad, but at some cost in power and area.
- 2=PINM_EXT_REG is the externalized pad control.
  This allows you to put a tiny pad control block closer to the pad, so the SCL_in to SDA_out time is reduced – does not affect SCL arrival to the peripheral. It is not normally needed unless the peripheral is a very long way from the pad (physically or logically) or you have really poor propagation times. An example of really long logical prop times would be a huge IO mux with a lot of depth and pins on either side of the chip, etc.

3. Details of the RTL

The block is divided into sub modules which handle base functions as explained in section 2.1.1. It further has multiple wrapper options, as explained in section 2.1.2, which controls how the block looks in the system.

This section explains how CDC (Clock Domain crossing) is handled, how the clocks work, how resets work, and the generalized states and inter-connects.

3.1 Clock Domain Crossing (CDC) solutions

Clock Domain crossing is an often misunderstood and over-engineered problem space (as a result). This section 1st explains the issues of both single-bit meta-stability as well as multi-bit meta-stability and re-convergence related to it. Then, it explains the approaches used in this block.

3.1.1 Single-bit meta-stability

For this explanation, the focus is on D type Flip-flops (DFF) – that is, dual latch (Master/Slave), with slave latch being exposed to D input on clock change, usually rising edge of clock; the same applies with an integrated enable DFF, often called an Enable Flip-flop (EFF).

Single-bit Meta-stability is the situation where the D input is changing inside the setup period of the Slave latch. The setup period is typically <= ~100ps in very small/modern process, and as large as ~800ps in very old process. The setup period is also variant with temperature, so changing the exposure time.

If the D is stable at the start of the setup, then the latch will correctly switch to the new state and settle very rapidly, giving the maximum Q output time for timing paths.

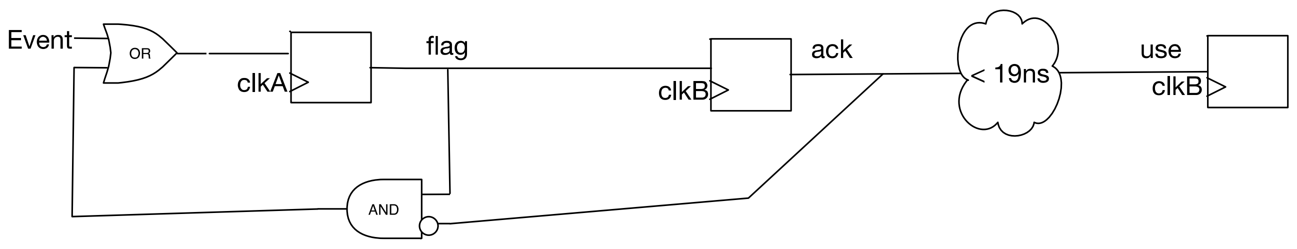
If the D changes during the setup time, the Slave latch may do one of 3 things:

1. Not change from its prior state – that is, not see the change. This is fine. It just means the change takes place on the next clock.
2. Change to the new state anyway and settle in the normal time. This is of course also fine.
3. Go to an unstable point for a while, possibly with some oscillations, where the two latch drivers and their feedback do not resolve quickly enough. This is the problem case.
   The latch will settle to one state or the other, but it may take longer than the “entitled” period given for path timing. So, if the Q runs over a longer distance or through a cloud of combinatorial logic, it is possible that the propagation of the settlement may occur after the next clock for the far end input. The most common and problematic case is that the Q output is split and run to different D input flops such that some of the flops get one value and some get another. This then leads to a combination of those flops’ states that is not supposed to be possible. For example, a state machine may be 3 bits and the 3 may change to a combination that is not legal. Such as Q→L→State[0]; Q→L→L→L→State[1]; Q→L→L→State[2]. If the settling time is enough for State[0] but not State [1] and State[2], you end up with a mixed state.

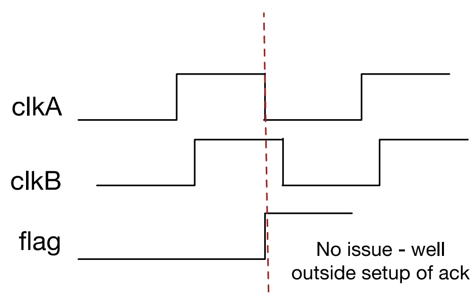
So, there are 3 considerations in Single-bit meta-stability:

- How long does it take the flop to settle? For modern process (180nm and smaller), the latches add “gravity” which means one latch or its feedback is weaker, so that the flop settles more quickly and certainly. This is even more so in 90nm and below, where it is normally less than 1ns settle time. In other older process, there is a a special sync flop which should be used.
- What is the distance (time) of the Q output to its use? This is why the traditional easy solution is just to feed the Q into the D of 2nd flop and so hide the problem. This is not the only way, but is simple. This works as long as the Q settles before the setup/clock of the 2nd flop.
- How many paths does the Q feed and could they be split by the timing? That is, if not just using a 2nd flop, what are the nets from the Q to all endpoints and are they all short enough. Further, if some are quite long, does it matter if they do not agree? (if they are independent, then it may not matter).

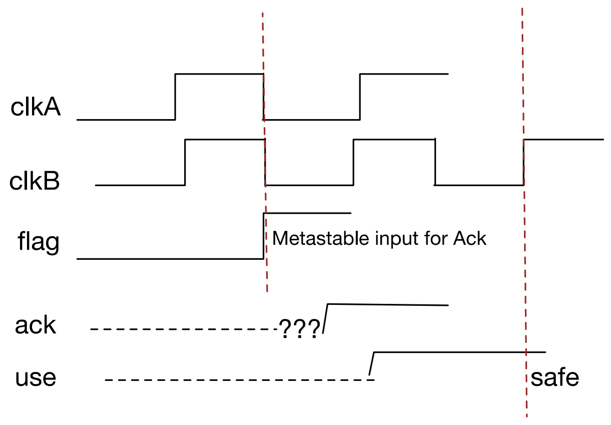
To understand the above, see **Error! Reference source not found.**.



A) shows an example setup with flag crossing domains. Two CDC points are AND(flag,~ack) and flag into ack



B) Shows non-metastable case where flag changes outside of setup of ack



C) Shows metastable case where flag changes inside setup of ack. Still safe if delay in cloud is less than metastable uncertainty of ack Q

Figure 4 - Showing single bit metastable event and how it can be handled safely

As shown in **Error! Reference source not found.**, the single bit metastable event occurs when a change occurs in the setup of a flop in a different clock domain. But, also as shown, there is no technical issue as long as the uncertainty (instability) is settled before the next clock. By definition, there must be a whole clock period of the receiving domain (clkB in this example). So, for example, if the cloud of logic and path uses 19ns and the period is say 25ns and the uncertainty is less than 5ns, then there is no issue with the feed into the “use” flop in the clkB domain.

In general, the model in the I3C block is to use single flops where practical, all embedded in SYNC_xxx modules (in sync_support.v). The specific implementation may swap in special sync flops if available and needed.

If needed, an implementation may choose to add a 2nd flop into the sync_support.v file for the SCL->CLK direction under the following conditions and reasons:

1. The system clock speed is very fast for the process.
    a. This means that it should not be used for the CLK->SCL direction since SCL is bounded at no faster than 12.5MHz.
    b. If the CLK speed is very fast, then SCL->CLK could use this since it will then turn around decisions fast enough to be safe. But, this should be considered.
2. The distance from a CLK side synchronized output in terms of actual distance and/or cloud of logic depth. If a higher depth, then an extra flop should be used.
    a. This can be easily determined by slack analysis from the net in question to its arrival points.
    b. It should be note that memory mapped register read rarely will be a critical path since the data is straight from the Qs. The long paths are normally related to RAM and NVMEM, and also to

"ready" nets.

3.1.2 Single bit synchronization

The other problem, independent of meta-stability is ensuring a change, and in particular a pulse, is propagated across a clock domain.

In general, the ratio of clock speeds between the two is not known and so this needs to work regardless of the ratio. Even if known, it is generally dangerous to rely on that for future uses. That is, if one clock is never more than 2x faster than the other, a simplistic model simply has the faster one count out 3 beats to be sure the other side sees it. But, if the rules change, the logic breaks.

There are 3 categories to consider:

1. A short duration pulse on one domain has to be reflected onto the other domain as a pulse.
   The input pulse may be 1 clock or it may be multiple clocks. The key point is that you cannot rely on the pulse lasting long enough to be seen on the other domain.
2. A level is passed (raw) from one clock domain to the other clock domain where it only matters that the receiving clock domain would see the level if it happened to want/need it. For example, a status bit (e.g. busy). So, the receiving clock domain does not care the state unless it so happens to want to check then (e.g. application software reads a STATUS memory mapped register).
   a. In that case, it needs to be registered for that use when it happens, but otherwise is ignored. The purpose of registering is so that the processor (or DMA or whatever) registering that bit is not dealing with metastable input (across the clock domain), only at most uncertainty of the Q output (of the registering flop); there is no specific need for "agreement" and the path to the register is extremely unlikely to be a critical path (since one layer deep of logic).
   b. But, if one worries about such things, an additional flop layer can be added, but as a result causing a stall of the bus.
3. A level is to be passed across the clock domain such that some action from the receiving side will change the level on the sending. This is a common occurrence but often not understood well by engineers reading the logic.
   An example, is request_x is sent from domain x to domain y. Then done_y is sent from domain y to domain x. Since the request may only be released upon seeing done_y change, the whole loop ensures that the clock ratios do not matter.

The most common 3 methods to deal with pulses and some level loops in this block are:

- 4-phase handshake. Y echoes X, and X does not change until Y has echoed it. This allows level loops and also allows pulse handling, as explained below.
- 2-phase handshake. Y echoes X, but only the difference of X and Y is used as the signal. This also allows pulse handling, but has some further advantages. This is used in some cases because SCL can stop suddenly, and so cannot be relied on to finish the sequence.
- Flag controlled data. This is explained in more detail below under multi-bit, but relies on the flag changing after the data (e.g. 1 clock later) so that the data is stable.

A 4-phase handshake is a way of ensuring that the receiving side has seen a change and copies the state over as well. For example:

- X changes from 0 to 1. This may be as a result of a "pulse" in X's domain or a level change.
- Y sees X is now 1, and it changes to 1.
- X sees Y is now 1, and it changes back to 0 but only if its input is also back to 0.
  - if X is based on a level, it just holds the level until both its input has changed and Y has echoed it.

        - If X is based on a pulse, say a 1 cycle pulse, it is goes to 1 due to the pulse, and holds 1 until Y echoes it.
- Y sees X is now 0, and it also changes back to 0.
    - Note that if this is about a 1 cycle pulse and the desire is for a 1 cycle pulse on Y, then a 2nd Y flop, say Y2, is used to reflect the change to 1. So, Y2 changes to 1 when Y is 1, and so the output pulse is when Y2==0 and Y==1, or (~Y2 & Y). That will last for 1 clock in Y's domain, regardless of how long before Y changes back to 0 (which cannot be faster than 1 clock).

A 2-phase handshake can be very useful for certain uses, but also presents some challenges and advantages:

- The 1st problem is that if the "difference" is X^Y (X and Y do not match), then that would be a metastable input to logic since X can change at any time.
    - So, 2-phase handshakes are usually used also with a 3rd flop.
    - The 3rd flop normally works exactly as with Y2 above. So, X changes, and Y sees X^Y==1 (knowing it is across the clock domain) and so Y flips. Now, in the next cycle, Y2 sees that Y2^Y==1 (same clock domain) and changes to match Y.
    - So, Y2^Y forms a 1 cycle pulse.
- The most popular reasons for using 2-phase handshakes are:
    - Faster since do not have to propagate the return to 0 each time. So, only phases 1 and 2 are used; this means 2 input pulses can happen closer together.
    - Works well if the sending side does not have a free running clock. In I3C, the SCL will stop on a STOP and so there may not be an SCL to support phase 3 (return to 0).

3.1.3 Multi-bit meta-stability and re-convergence

Multi-bit meta-stability builds on the single-bit case to add another wrinkle. There are 3 common cases, although they are identical in reality, they differ only in how the logic is used:

1. A multi-bit vector is registered from another clock domain. So, x[7:0] <= y[7:0] is an example. The risk when from different clock domains is that y may be changing at the same time as x is registering (copying) the Q outputs. Ignoring single-bit meta-stability, the issue is that for example, y[3] and y[7:6] may happen to change 400ps sooner than y[2:0] and y[4], and that may be 200ps before y[5]. That is fine for other uses in y's domain since the timing would never allow it to be within that 600ps. But, x's domain is asynchronous, so x may happen to be registering when y is changing and so x may for example copy the new y[3] and y[7:6] but get the old y[2:0], y[5:4]. This is likely not acceptable to the use.
2. One domain wants to use the data directly from another domain, and so the receiving side needs to know when the data is stable and safe to avoid both the general problem in 1 above, but also the possible split of data through different path lengths.
3. Two different paths, such as "enable" and "request" may be changed at the same time in one domain, but may be picked up incorrectly in the other. For example, enable_x <= enable_y; req_x <= req_y; Then, enable may get picked up as changed to 1, but request is picked up as its old value.
   This case is the same problem as the x and y case from above, except it is (a) harder to recognize than a vector, and (b) even more likely to have separated timings because the tool does not see them as directly related.
   This case is usually referred to as "re-convergence" since the relationship of enable and request on side is converged back to a similar relationship on the other.

3.1.4 Solutions to the CDC issues related to single-bit uses and some multi-bit

The choice of solution depends on the needs. This block employs different methods for different types of data:

- For raw status bits to be shown to the application when it reads the STATUS register, the independent bits are registered in the 1st cycle of the APB read operation.
- For from-bus (from master to us), the SDR engine has two 8-bit buffers to write into. The system side (clock) controls the selector for which buffer to write into. This uses a gray code 2-entry FIFO model such that the writer (SCL side) can write as long as not full, and the reader can read as long as not empty. The gray code model means that both bits synchronize independently since only 1 can change at any one time.
- For to-bus (from us to master), the SDR engine sees two 8-bit buffers read from. The system side (clock) controls the selector for which buffer and it also owns the buffers. Again, a gray code model is used, such that the SCL side can read when not empty and the system side can push when not full.
- For data errors, 2-phase handshakes are used since SCL can stop suddenly.
- For “events” to be copied into the system domain, such as START, Header matched, etc, a 2-phase synch is used with an extra Y2 type flop. This ensures safety and works well even if SCL is stopped on the next half SCL cycle (when SCL is High).
- In general, most things are handled from the system domain, since SCL is not free running. In some systems, SCL is faster than the system clock (e.g. bus clock), but that is OK for this use model. Also note that SCL rising is used in some cases, which is ½ clock before or after the falling edge event.

3.1.5 Solutions to the CDC issues related to some multi-bit and re-convergence.

The most common solution used for multi-bit cases as above is to have a “valid” signal change after the multiple bits change. So, for example, if Y[7:0] is changing, then a signal called Y_valid is pulsed to say that Y can be registered/copied, but done after the data has settled (1/2 or 1 clock later).

Note that if Y_valid changes in the same cycle as Y[7:0] and the X side just uses it directly, you would have the same re-convergence AND multi-bit problem (since the order of Y_valid and Y[7:0] changing is not controlled.

So, there are 2 choices:

1. Y_valid changes 1 cycle later or ½ cycle later in the Y domain. So, by the time X sees Y_valid change, Y[7:0] is fully settled. If Y_valid is only used to register Y[7:0] into X[7:0], this is all that is needed.
2. Y_valid changes at same time as Y[7:0], but a 2-phase or 4-phase handshake and sync is used with a X2 on Y_valid, so that the actions are from X_valid^X2_valid (2-phase) or X_valid&~X2_valid (4-phase). This works because it then waits 1 cycle on the X domain to register Y[7:0] into X[7:0], when it is stable.

The choice of method usually depends on which clock is assumed to be faster and (since waiting a clock) or which is free-running (since the other one may not be able to rely on extra clock to delay).

The re-convergence cases, such as the enable/request example work the same way; the simplest is to have “enable” change 1 clock to ½ clock after in one domain or the other. That way, the “request” is then stable.

3.2 Clocks, flops, and states

The model is somewhat complex because we choose to register SDA vs. use it combinatorial. The reason for registering is that there is some risk of SDA changing into SCL and this will create more of an issue with metastable inputs running through multiple nets. Further, it is modal (or state based) with i2c and I3C SDR forming one mode, HDR-DDR forming a related mode, and HDR-TSP/TSL forming an unrelated mode.

3.2.1 I3C SDR and i2c mode

The model has 4 “clocks” for i2c and I3C SDR:

- SCL falling:
  - Handles the START detect (including repeated START) as well as START error (SCL falling from STOP but with SDA still High).
  - Drives the state machine. State is partly combinatorial on START since start comes out of the blue (we have no clocks between stop and start).
  - Completes each Data byte (with ACK or T to follow, as explained below) and all other main SDR controlling logic.
- SCL rising:
  - Registers SDA
  - Handles 1st START (init) which is from an implicit STOP.
- SDA falling:
  - Only used in the HDR Exit and HDR Restart detector. Not used for SDR itself.
- SDA rising:
  - Is the STOP bit when SCL is High. This has no corresponding SCL. A START will eventually follow before an SCL. This is reset by the START in the SCL falling domain.

The four clocks are formed from a combination safe MUX and XOR (or MUXed INV), which is disabled when DFT so that all clocks are from a DFT_clock.

Note that in HDR-DDR the SDA driven clocks are not used and the HDR-Exit pattern detector is used instead (which is SDA and SDA_n driven). Likewise if HDR-TSP or HDR-TSL, then the whole SDR/DDR block is held off and only that block is used, along with the HDR-Exit pattern detector. See details below.

There are 4 resets:

- Global reset fed into block. Everything is reset by this global reset. Release is assumed synchronized to the system clock (if provided). It is not synchronized to SCL/SDA since those should not be active when the block is released (or it would be in Hot-Join and so not active).
- STOP is reset by START as well.
- The DAA block is reset when not in DAA mode.
- The CCC state detector is reset by STOP. This is because no clock accompanies STOP.
- HDR Exit and Restart detector uses special resets.
- HDR modes are held in reset and clock gated by not_HDR which is set by Exit Pattern and cleared by EnterHDR.

3.2.2 HDR-DDR Mode

The HDR-DDR mode uses the same clock model as the SDR mode. It simply has to manage the 2 bits per SCL clock, using rising and falling. To be compatible, the same SDR_n (falling edge) model is used, so rising is reserved for registering data.

3.2.3 HDR-TSP and HDR-TSL Modes

HDR Ternary modes have to extract a clock from the data for inbound data and have to generate their own clock for outbound (Slave read from Master). The high speed clock is provided by the system, but clock gating is supported by the block indicating when the high speed clock is needed.

3.2.4 Slow Clock and IBI for "Bus Available"

IBI is handled in one of two ways:

- If the Master initiates a START, the Slave will emit its address if an IBI (or MR or HJ) is pending.
- If the I3C bus is in the Bus Free condition for greater or equal to 1µs, then the Slave will pull the SDA

Low to force a START from the Master.

For the latter case, the Slave needs a clock to generate the ~1µs timing. To support this, a Slow clock (CLK_SLOW) can be provided to save on power. Additionally, the block provide a slow clock gate (slow_gate) which is 1 if the CLK_SLOW can be turned off.

The system has 3 basic choices:

1. Use the same clock as CLK, which may be variant. If variant, the slow match register can be used to change the match rule.
2. Provide a truly slow clock, which makes it inexpensive in power to count 1µs or more.
3. Provide some other clock, but use the slow_gate signal to gate the clock near its source.

This model is used in conjunction with the parameters for width of counter and match rule.

3.2.5 Protocol Modes and States

The following modes are activated in i3c:

- i2c mode is the default mode on startup. If there is no i2c static-address, this has no effect other than to track frames looking for I3C transitional frames.
    - If a static address is used, the device can interact with the i2c bus master using it.
- While in i2c mode, an i3c transitory frame mode occurs whenever the I3C Broadcast address is seen. In particular:
    - 7'h7E is broadcast and then ENTDASA from Master.
        - If we match our static address (if any) or 7'h01, we will be assigned a Dynamic Address and enter I3C SDR mode.
    - 7'h7E is broadcast and then ENTDAA from Master.
        - If we are able to get our 48b ID out and are assigned a Dynamic Address, we will enter I3C SDR mode.
    - If we generate a Hot-Join arbitrated event (sending 7'h02 when Master generates some other address such as 7'h7E), there is a conceptual Hot-Join mode.
        - This will be resolved by ENTDAA or ENTDASA setting a Dynamic Address and so entering us into I3C SDR mode.
    - Note that an I3C Slave can operate as a normal i2c slave only if it has a static address; otherwise it matches nothing in i2c and waits for the above only.
- I3C SDR mode is the standard mode once I3C has been activated, which is defined by the Dynamic Address being assigned. This is the resting mode of I3C and all devices are normally in SDR.
    - It is exited by the RSTDAA CCC, which returns to i2c mode. That is not normally used.
    - Note that use of SETNEWDA does not affect the mode, only our Dynamic Address.
    - Note that once in this mode, the device will ignore messages to/from its original i2c static address.
- I3C CCC command sub-mode of type Direct or Broadcast.
    - The Broadcast CCC sub-mode is exited by a repeated START or STOP.
    - The Direct CCC sub-mode is exited by a 0x7E broadcast address after repeated START or by a STOP.
- I3C DAA CCC command enters the DAA mode. This is a special mode for dynamic address assignment and is exited by STOP.
    - It is ignored once the Slave has a Dynamic address.
- I3C SETDASA CCC command enters the static address match sub-mode. This allows matching the

device's static address (if any).

  - It is ignored once the Slave has a Dynamic address, and is ignored if the Slave in i2c mode has no static address
  - Note that the special point-to-point address is also matched.
- I3C HDR modes are activated by an ENTHDRn CCC commands with the n value being the type of HDR (0 to 7). This is valid until HDR Exit Pattern, whether supported or not.
  - Includes DDR, TSP/TSL
  - Exit Pattern detection may be left on at all times or only activated on ENTHDR. It should be left on at all times to prevent errors.
- Internal States also exist, such as IBI/no-IBI, Low power mode, etc. These are flagged internally.
  - They are exported to app and also affect the engine so it does not perform a forbidden operation.

3.2.6 Address match

The Slave will match 2 of 3 possible addresses, depending on internal state:

- 0b111_1110 (written as 7'h7E in spec), which is the I3C broadcast address.
- i2c style static address, but only if:
  - The device has a static address
  - Is not in I3C mode. Will only match until ENTDAA or SETDASA command has assigned a Dynamic Address. Note: SETDASA matches the static address or special point to point magic address.
- I3C Dynamic address, once assigned by ENTDAA or SETDASA or modified by SETNEWDA.

The Address match is passive (just listen) for all repeated START and also for START, unless an IBI/MR/HotJoin has been activated. If passive, it simply tries to match. If not matching, it waits for repeated START or STOP. If an IBI/MR/HotJoin, the arbitration mechanism is used for START (but never repeated START).

If matching the 0x7E broadcast address, it will listen for a CCC until next repeated START or STOP. The CCC_possible bit will be set and cleared for this.

3.2.7 Slave: Data Write from Master

When the address match was valid for the Slave Dynamic address, and the type was W (write from master), the slave will ACK the address, unless some condition prevents it such as a full buffer. On ACK, the WRITE state is entered. Once ACKed, it waits for each complete data byte. This is handled in two steps:

- 8 bits clocked in and stored in the next buffer location while in the WRITE state.
- I3C: On the 9th bit, the W9TH state is used and the parity is checked. The buffer is marked as complete with or without a parity error.
- I2c: on the 9th but, the W9TH state is used and the data is ACKed and stored.

When the address match was valid for 0x7E broadcast address, the Slave will ACK it and set the in_ccc Possible flag. On 1st data complete, it will set the in_ccc Broadcast or Direct flag (based on bit 7 of cmd). It will then parse the command if recognized, setting that bit as well. The recognized ones are for Dynamic Address work and modes; the CCC and DAA blocks handle the work load after that. If not supported, then they will be passed up to the system.

3.2.8 Slave: Data Reply to Master

When the address match was valid for the Slave Dynamic address, and the type was R (read by master), the slave will ACK the address, unless there is no data waiting for the read. On ACK, the READ state is entered. Once ACKed, it will emit the data byte at a time, with the 9th bit using R9TH state. If I3C, the T bit is emitted to allow the device to indicate when last byte (from input signal from upper layers) or not last byte. If not last byte, the Master may terminate the read via the T bit. If i2c, the 9th bit allows the master to terminate via NACK, else allow us to continue via ACK.

On completion of each byte read, the done signal is pulsed to get the next byte.

The application is notified if the read was terminated by the Master unexpectedly (abort in i3c, NACK before END marked).

3.2.9 Start and Stop appearing in different place

Repeated START and STOP can occur in the middle of data or address. It is not uncommon to be in the 1st data bit, since that allows the master to setup the SDA in the right way. But, it can also occur at other times. The state machine and modes all have to be able to accommodate. This is more complex than it may appear because STOP is not clocked by SCL. As a result, the state machine uses a combinatorial version of “state” which is affected by the non-clocked STOP (which means START) as well as handling a STOP and then SCL change without a proper START.

3.2.10 Slave Bus Idle detect and use

If the application wants to perform an IBI or Hot-Join or MR, it normally will wait until a normal START following a Bus free condition. However, if a longer time goes by after a STOP, it is an IDLE and the logic can drive an SDA low, releasing on SCL low. This is signaled by upper layers using the force_sda signal.

The force_SDA (force_START) is handled with a counter using the system clock. Parameters indicate counter size and match. The SCL side changes its value to match the system side whenever there is an SCL clock, so the counter resets (and maybe is no longer needed if the IBI gets out). If needed, the force_START is still protected on the SCL side such that it will not act combinatorially unless still in STOP state.

3.3 Resets and use of Layered Resets

The I3C peripheral uses mostly a single external global reset (ie. RSTn and/or PRESETn). This is assumed to be held Low until the block is allowed to be active, in which case it is released such that the block is not concerned with clock edge sync; it is not concerned with clock edge sync because flops are not active by themselves. If the global reset is used for a system or peripheral reset, it should be held Low long enough for the reset to propagate to all flops in the peripheral.

3.3.1 Local layered resets

In addition to the global reset, some flops will use a local layered (aka hierarchical) reset. This means a locally produced reset control will be used in addition, such that:

1. Global reset will always reset no matter what
2. The local reset can be disabled by a Scan based disable: scan_no_rst
3. The local reset will be in the same clock domain as the flop, so it will release after one clock edge and well before the next clock edge for the flop **or** it will be such that the D input is not changing on its release, so clock sync is not needed.

The local reset is generally used due to the nature of the SCL clock stopping without warning. Such a use prevents the case of states continuing past a STOP (or EXIT-STOP in HDR). This also controls the entry states when needed – ie. flops/states will be held in reset until it is safe for them to operate.

The local reset expression normally looks like:

```
assign my_reset_n = RSTn & (local_reset_n | scan_no_rst);
always @ (posedge CLK or negedge my_reset_n)
  if (!my_reset_n)
    my_flop <= 1’b0; // reset
  else if (...)
```

. . .

This matches the rules given above. The local_reset_n term will generally be minimally combinatorial wire or expression, such as ~some_flop or |some_flops, such that it forms a single continuous hold of reset, with a release under some conditions. That is, its typical use is to hold the flops in reset until some safe condition. So, its purpose will often be to exit the state even when the clocks stop suddenly, but also to enter only when appropriate.

It is notable that "glitches" (noisy transitions) on the combinatorial logic are not considered a concern in the sense that there are only two cases for this single state reset model:

- The entry to reset glitches, which is OK because the end result is a hold in reset for a prolonged period.
- The exit glitches, which is OK because the non-reset state only matters once the next clock edge occurs. The local reset source in almost all cases will be from the same clock domain, so will change (with or without glitches) just after a clock edge, and so long before the synchronized release takes place (on next edge).

3.4 Data handling with and without FIFO

The low level slave engine is fed a buffer for to-bus, and provides its own buffer for from-bus (so bit insertions do not build a deep mux into a FIFO). This is described below in sub-section 3.4.1

The full wrapper and so APB interface allows for 3 methods:

- Ping-pong buffers (2 entry FIFOs) for each of from-bus and to-bus only, as described in sub-section 3.4.2.
    - This interfaces to the APB register model, if used, using the byte read and byte write regs, along with interrupts.
    - If not used, these form the buffering and clock-domain-crossing (CDC) interface for direct nets and external FIFOs.
    - The from-bus one ends up being a 3 entry buffer (but 2 entry FIFO) since it has a holding buffer. The holding buffer in the CLK domain is to keep the path short for an external FIFO (no mux) and to give a bit more time for a processor ISR.
- Deeper than 2 entry internal FIFO, as described in sub-section 3.4.3.
    - These replace the ping-pong buffers with deeper FIFOs which form the clock-domain-crossing (CDC) interface as well as deeper buffering.
    - This allows for lower speed system clocks without having to slow the i3c bus rate.
    - It is possible to engage for only one direction (e.g. from-bus).
- External FIFO with works with the ping-pong buffers, as described in sub-section 3.4.4.
    - The external FIFO is operating in the system clock domain and moves data to/from the ping-pong buffers byte at a time. So, the clock-domain-crossing (CDC) is still in the ping-pong buffers.

3.4.1 Slave engine data handling

The low level slave engine (i3c_sdr_slave_engine) is designed to process byte at a time, except when HDR-DDR is enabled, when it stores 2 bytes for from-bus, waiting on parity.

The slave engine creates the register(s) for from-bus, including an extra byte register to handle CCCs (for inspection uses). The layer above the engine indicates buffer status for from-bus as a handshake (in the same clock domain) when it copies up the data (into the ping-pong buffer or internal FIFO). If the buffer is not available when needed, the engine signals the error.

The slave engine uses only inbound nets for to-bus. The layer above the engine provides a byte and flag and the state of that byte (empty or full); the flag indicates if last byte (end of data). If the data is not available when needed, it will signal an error. There are 3 errors that can come from to-bus:

- No data available when ACK/NACK, so NACKed.
- No data available during read and was not END in a previous byte. So, FF is sent.
- Master terminated before END. In i2c, this is NACK before END. In i3c, this is abort.

3.4.2 Ping-pong buffers (aka 2-entry FIFOs)

The full wrapper instantiates i3c_data_frombus and i3c_data_tobus to handle data buffering and clock crossing, whether a bigger FIFO is used or not. These also handle errors crossing over the clock domain.

The ping-pong buffering is managed as a 2 entry FIFO – one for from-bus and one for to-bus. These use gray codes (2 bit) to support clock domain crossing (CDC) and indexing into each of the 2 buffers.

The 2 entry scheme is handled differently than larger FIFOs because it is a simpler model. The handling is done by gray code only, since gray to binary is just XOR of the 2 bits, and counting in gray is easy.

Note that from-bus copies the slave engine buffer up into the FIFO – this is done so that the engine is not bit managing the FIFO, which causes larger muxes. Further, from-bus has the equivalent of 3 FIFO entries, using a separate holding buffer. This means a bit more area and power, but protects the path for external FIFOs.

The to-bus buffer model uses 9 bits so that the END marker is preserved with the data. This is passed into the engine as tb_end when it sees the byte.

3.4.3 Internal FIFO

The internal FIFO uses the Sunburst design approach of CDC. It uses a binary counter and then a binary-to-gray converter. This allows for sizes as a power of 2, including 4, 8, 16, etc entries. The FIFO itself is instantiated as RTL “memory”, which works well to about a 16 entry FIFO. The bigger growth is not the FIFO memory, but the muxes into it as the size increases.

The FIFOs replace the ping-pong buffers when enabled. It is possible to enable only one direction and/or to use different sizes between from-bus and to-bus.

The FIFOs also provide a gray to binary reconvert to calculate number of entries. The number of entries count is used for triggering interrupts and to allow the app to read the number of entries.

When this is used, the register interface (when APB wrapper is used) supports the FIFO management.

3.4.4 External FIFO

The full wrapper supports 2 methods for external FIFO. This allows for shared FIFO and also for RAM based or other specializations. The FIFO in this case will be in the system clock domain, so clock crossing is handled by the ping-pong buffer as described earlier.

The external FIFO does need to support the “END” marker.

The model is a simple handshake as shown below.

The to-bus (TXFIFO) sets avail when a byte waiting to go out, and the peripheral will pulse used for one PCLK when taken. The “last” signal indicates the last byte (END). The peripheral will signal start when it sees a START and data is available.

```
    // To-bus means from slave to master
  input           ixf_tb_avail,     // 1 if byte is available for tobus
  input           ixf_tb_last,      // byte is last for message
  input     [7:0] ixf_tb_data,      // actual data when avail=1
  output          oxf_tb_start,     // pulsed when ixf_tb_avail==1 and START
  output          oxf_tb_used,      // pulsed when tb_data used
```

The from-bus (RXFIFO) is likewise a handshake. The FIFO sets free to 1 to show it can take another byte. If free=1, and data is available from the bus, the req port will pulse 1 PCLK to have the FIFO accept the next byte. The eof port will indicate that the frame ended in repeated START or STOP..

```
    // From-bus means from master into slave
  input            ixf_fb_free,       // 1 if space to take a byte frombus
  output           oxf_fb_req,        // pulse when data to be taken (and free=1)
  output     [7:0] oxf_fb_data,       // data frombus when req=1
  output           oxf_fb_eof,        // frame ended in repeated START or STOP
```

3.5 DMA (for internal FIFO use only)

The DMA model supports a few types of DMA uses:

- Normal Request/Ack: The peripheral holds Request=1 while RXFIFO has data or TXFIFO has space.
    - DMA_TYPE=0 is used if ACK is before the access to the WDATA/RDATA MMR. This ensures that the Request is dropped if last byte/half-word.
    - DMA_TYPE=2 is used if ACK is with or after the MMR access, or if ACK=0 always. In that case, the Request is 1 while more data to process.
- 4-phase Handshake between Request/ACK.
    - DMA_TYPE=3 is used if 4-phase handshake is used, and ACK is with or after the MMR request. The 4-phase model de-asserts Request each time ACK goes High, and then re-asserts once it is Low and there is more data to process.
- Other types are not supported such as Trigger-burst, 4-phase before the MMR access, etc.

3.6 Slave Reset and RSTACT CCC

The Slave Reset mechanism is designed to allow an always-on tiny block to monitor SDA and SCL for a specific pattern based on HDR Exit – extended with repeated START and STOP at end. This detector can be used to wake or reset the device or just peripheral, as well as do nothing for any specific reset. The normal use is that the Master emits a RSTACT CCC message saying not to reset when this pattern is emitted just after; devices which are broken will reset (since they miss the CCC) and devices in deepest sleep (e.g. unpowered core domain) will wake (since they also miss the CCC).

The 1st default action is only to reset the peripheral. If that does not work, the next time will reset the chip/system.

The Master may also request a specific action, which the slave can dial in if it supports that (e.g. reset the peripheral only).

The Slave Reset is connected into the system as shown in Figure 5. Note that it is outside of the I3C Peripheral, and may be in an always-on domain. Sleep/wake signals are not shown. The common connections are as follows:

Reset Detector	**Connect to IP**	**Connect to System**	**Notes**
I3c_slave_active	raw_slvr_isslave if master+slave	If I3C can be unused	Connect to enable if can be unused – should probably not clear for safety reasons.
iRstAction[3:0]	raw_slvr_reset	-	Result of RSTACT CCC
oRstRstAction	iraw_rst_slavr_reset	-	Clear RSACT state

oResetBlock	iraw_slvr_irq	-	Triggers IRQ in IP
oRstAll	-	System reset	Should perform similar to pin RSTn
oRstCustom	-	Custom Reset if enabled	Must be setup via RSTACT_CONFIG[2]
iDeepestSleep	-	Feed from system or 0	Only used if deepest sleep supported
oWake	-	Wake controller	“

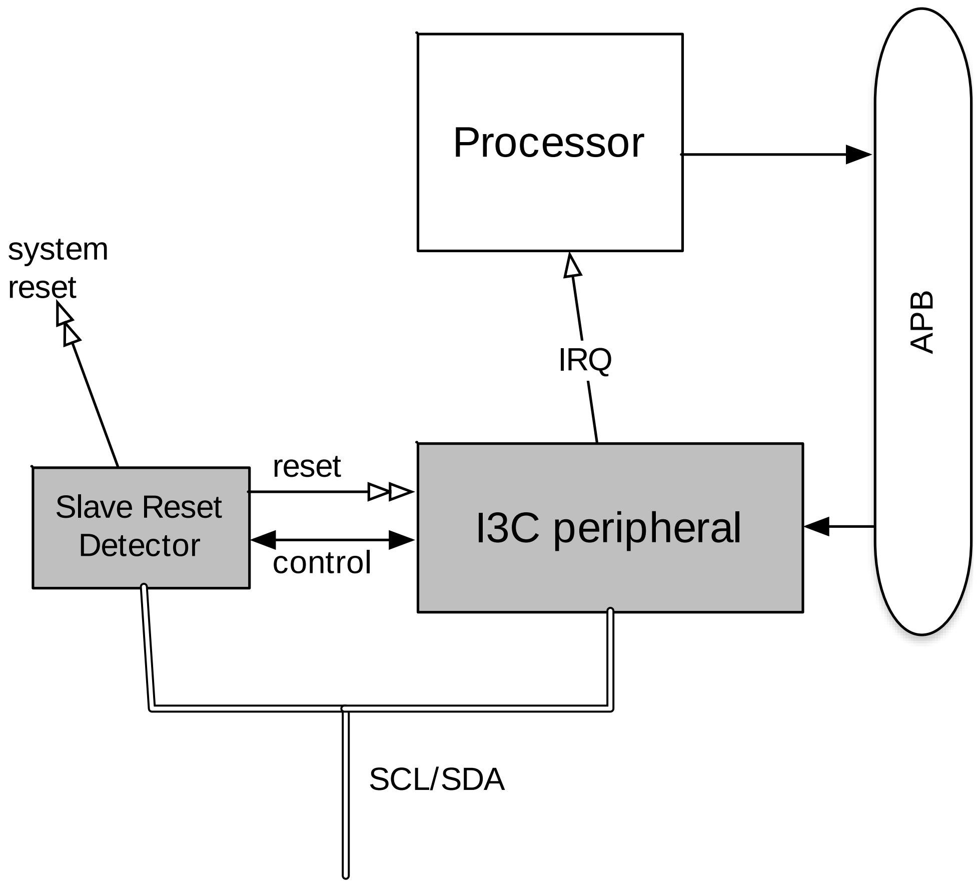


Figure 5. Showa Slave Reset Detector connected in the system (Slave Reset may be in Always-on domain).

Figure 6 shows the connections of the Slave Reset detector, including support for deepest sleep and wake. If those are not used, they are just not connected. If the I3C peripheral may be powered off when the reset detector is powered, the isolation cells are used to protect it, although the nets are logically gated off when iDeepestSleep is 1.

Note that the block wants the SCL and SDA to be fed as clocks. This is done outside the block, using any method needed (e.g. using the CLOCK_xxx type blocks). 3 domains are needed to support the reset.

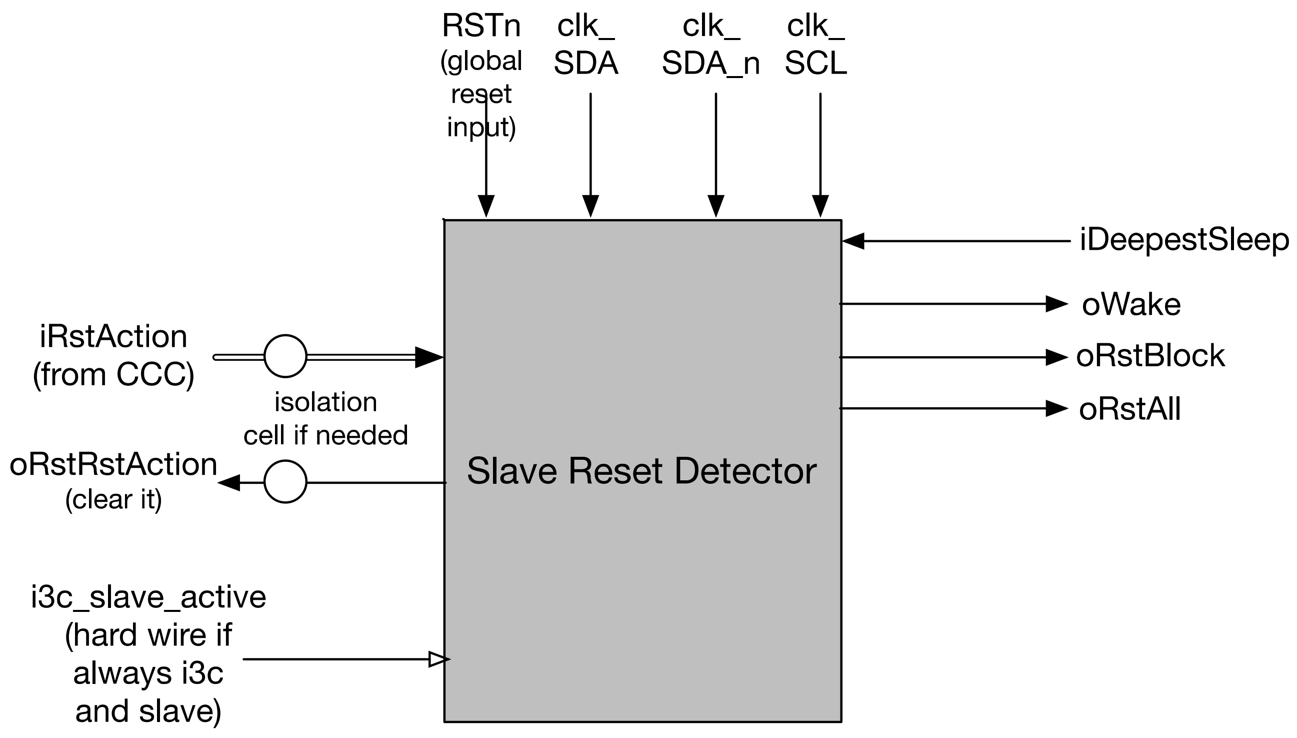


Figure 6. Showing reset detector connections – not all will be used in many systems

The i3c_slave_active signal must be used with caution, as it will disable the reset detector when 0; it should be strapped 1 if always i3c and slave, else should be safely controlled (or may defeat the purpose of the Slave Reset block). If the i3c peripheral is optional, then it must be selected by some system control feature – that should be what controls i3c_slave_active.

The block is connected to the i3c peripheral via the iRstAction and oRstRsttAction ports. These will be clamped off if iDeepestSleep signal is 1; if that kind of sleep is not possible, the iDeepestSleep port should be strapped 0. The 2 nets can be treated as asynchronous even though from the same SCL. This is to avoid having to do setup/hold timing between the modules when the reset detector is in a separate domain. That is, the clk_SCL and clk_SDA_n are not intended to be treated the same clocks as used in the peripheral – this allows freedom in placement of the detector.

Finally, the reset and wake signals are connected to the system; they will be 0 when inactive, 1 when actively driven. There are clearing events used to clear: RstAll waits for the Reset input to assert (or i3c_active to go to 0); the RstBlock will wait for the I3C peripheral to clear its cause register or to see a RSTACT or GETSTATUS. Wake will clear on release of iDeepestSleep.

3.7 Slave FSM

The Slave side FSM processes i2c and i3c SDR. It is IDLEd during HDR and DAA modes.

Note that STOP is not a clocked state (since SCL is already high) and so START is recognized because of preceding STOP.

If SDA changes High when SCL is high (STOP), we abandon state in terms of next_state, but cannot affect state itself. If SDA changes Low when SCL is high, we move to A7 for match only (cannot arb since repeated START).

We register SDA on rising edge but act on live SDA on rising edge for i2c and SDR mode. We must write SDA on falling edge, including special case of SDR mode read T bit, where we also tri-state on rising edge.

State	Next	Actions	Notes
IDLE	A7 when SCL goes Low and STOP registered.	May drive SDA Low for IBI if bus-free. May drive SDA Low for Hot-Join if bus-idle.	Can only get there due to STOP having been registered using SDA rising edge, and the START from SDA falling edge.
A7-A1	Next one on each SCL rising edge to RnW (as A0 equiv).	We match if no IBI, MR, P2P, or Hot-Join. Else, we inject by arbitration.	If we are injecting and we want 1 and we see SDA Low on SCL rising, we abandon; no match since we would emit DynAddr and 0x7E always loses to us.
RnW	If we have won arbitration so far, then depends on if we win last bit (we must if W, else could lose on R). If no match, we go to WAIT to wait for repeated START or STOP. If match (our DynAddr or 0x7E), we go ACK_NACK.	We finish arb if injecting for IBI, MR, P2P, or Hot-Join.	
ACK_NACK	If ACK, move to READ or WRITE. If NACK, then move to WAIT.	We have matched, but we can NACK if unfavorable from external controller.	Should not be NACKing normally.
READ	Stays in READ for 8 beats per byte, then READ_T special handling if not last byte, else READ_END if last.		Read terminator bit is special for rising edge of SCL by using Hi-Z.
WRITE	Stays in WRITE for 9 beats per byte. If parity fails, or external controller signals abort, then moves to WAIT.	Compute Parity on 9th bit. Pack the bytes into words.	
READ_T	Moves to READ if not aborted by Master.		Stays in Hi-Z
READ_END	Moves to WAIT.		
WAIT	Stays in WAIT until STOP or START detected. Note that cannot really leave WAIT on STOP since no clock. So, real state is state & ~is_stopped.		

3.8 Debug Observer mechanism

The debug observer model allows the integrator to add observation nets to bring to the outside. The integration can then apply any muxing to the desired width/pages.

The model uses include files. The default include files are stubs and define all ports as 1 bit wide and just pass the 1’b0 up the chain (from i3c_sdr_engine up through i3c_slave_wrapper and then i3c_full_wrapper and then to the outside wrapper (slave or master).

The outside wrapper allows defining the parameter TOP_DBG_MX with the width of the observer minus 1.

The outside wrapper then outputs out_debug_observ as a port which is defined as TOP_DBG_MX:0 as a vector. Each include file can build up the bits that feed it.

The include files are:

- Dbg_observ_top.v: included by top level wrapper. Also defines the full wrapper port
- Dbg_observ_full.v: included by the full wrapper. Also defines the slave wrapper port
- Dbg_observ_slv.v: included by the slave wrapper. Also defines the SDR engine port.
- Dbg_observ_eng.v: including by the SDR engine.

4. Wrapper details

The overall peripheral is wrapped by one of 2 or 3 possible wrappers. These are divided into:

- APB with built in registers matching the Programmer’s model spec, with specific memory mapped registers controlled by parameters.
- Full system but without an APB bus or memory mapped registers.
- Minimal system without APB bus or memory mapped registers.

Note: the autonomous wrapper over the minimal system is separately defined.

The wrappers minimally perform clock domain crossing work, data mapping, and work around the instantiation of the Slave.

4.1 Common ports

Ports into and out of the wrappers are based in part on parameters and what the wrapper does. The common port model outlines the naming convention and the role of the ports. Note that some may not make it out of the interface for APB and minimal non-APB.

4.1.1 Configuration inputs into SCL domain – no CDC

There are a set of nets whose job is to support the parameterization. These are unchanging at runtime and so no special CDC work is done. These all start with **cf_** and input is shown in terms of the block’s view. Most are only used when enabled by parameter:

```
// 1st two are not necessarily held, but are used when bus is not active
input           cf_SlvEna,        // Enable signal – may be strapped 1
input           cf_SlvNack,       // Temp use: Slave should NACK all messages if 1
// next set are related to addressing
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
input     [7:0] cf_SlvSA,         // Static address for i2c if [0]==1 (and enabled)
// more such nets whose use depends on params (CCC, HDR, etc) being enable
input    [11:0] cf_MaxRd,         // Max read return (if CCC enabled)
input    [11:0] cf_MaxWr,         // Max write for master (if CCC enabled)
input           cf_DdrOK,         // Allow DDR messages (if DDR enabled)
input           cf_TspOK,         // Allow TSP messages (if TSP enabled)
input           cf_TslOK,         // Allow TSL messages (if TSL enabled)
input           cf_BAMatch,       // Bus Available count match (if IBI and dyn BAM)
```

4.1.2 Raw states back from SCL domain – with or without CDC as needed

The following raw states are used to allow the SCL domain to provide status information, events, and states. If used by the system, it would be registered before actual use. For example, a STATUS memory mapped register would register the raw value before returning (e.g. on 1st cycle of a bus request).

The raw nets all start with **raw_** and output from the block that way. The ActState and EvState are I3C bus states as set by the Master, when the CCC handling is set to handle these.

```
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)
output   [11:0] raw_timec_sync,   // Time Control sync data. [11]=chg, [10:8]=cmd
```

The interrupt nets all start with **int_** and are usually synchronized before being output (see details below under the full wrapper); they are mostly synchronized and grouped into “interrupt” style bits and related ones for errors. So, more likely to see as outp_IntStates, outp_GenErr, outp_DataErr with corresponding clear controls.

```
output          int_start_seen,   // 1 cycle pulse when START or repeated START seen
output          int_start_err,    // held when SDA=1 after STOP - cleared on START
output          int_da_matched,   // 1 cycle pulse if matched our dynamic address
output          int_sa_matched,   // 1 cycle pulse if matched static address (if any)
output          int_7e_matched,   // 1 cycle pulse if matched i3c broadcast address
output          int_ddr_matched,  // 1 cycle pulse if matched our DA with DDR
output          int_in_STOP,      // STOP state (held until START)
output          int_event_sent,   // 1 cycle pulse when event (event_pending) out
output          int_event_ack,    // 1 cycle pulse with int_event_sent if ACK vs. NACK
output          int_in_evproc,    // 1 if processing IBI now
output          int_ccc_handled,  // held if CCC handled by us
output          int_ccc,          // 1 cycle pulse if CCC not handled by us
```

The state nets all start with **state_** and output from the block that way (raw). The state_req set are a live *bus busy* informational that includes what is happening now, as explained in the STATUS register’s bits [6:0].

Note that some are always 0 if not selected by parameterization:

```
output          state_in_ccc,     // held while true - context
output          state_in_daa,     // held while true - informative
output          state_in_hdr,     // held while true - informative
output    [6:0] state_req,        // [6]=HDR, [5]=ENTDAA, [4]=Rd1st, [3]=Rd,
                                  // [2]=CCChandle, [1]=W/R, [0]=busy
output    [1:0] opt_state_AS,     // same as raw_ActState
output    [1:0] opt_ev_mask,      // same as raw_EvState
```

4.1.3 From bus (Master to slave) data and handshakes and errors

The set of nets related to data from the Master is modified based on the wrapper. The 1st step is synchronization and the 2nd is interfacing to a data port or FIFO. These details are covered in more detail in the wrapper.

The critical role of the frombus and tobus blocks is to handle clock domain crossing and to hide most of the details related to timing and FIFO.

These blocks have their own optional holding buffers and minimally a 2 entry CDC ping-pong FIFO which can be replaced with a larger internal FIFO by param. If an external FIFO is to be used, the 2 entry ping-pong FIFO is used to decouple the CDC. The SDR engine also holds its own SCL domain 8b buffer for from bus, plus a private 8b buffer for CCCs handled by the block. The tobus block must also get an end marker from the System to know when the data is finished.

In addition to managing data buffers and optionally a FIFO, they handle errors of type over-run, under-run, under-run with NACK of header, termination of read by master, and parity errors. These are also synchronized.

4.2 APB Wrapper details

The APB Wrapper uses the memory mapped registers to handle most details, as explained in the Programmer's model specification. This may include:

- Configurations that are not handled by the parameter constants, such as 48 bit ID in whole or part, BCR (except where modified by other details), DCR, i3c static address, Max limits (if used).
- Control to launch events such as IBI, MR, and Hot-Join.
- Status and interrupts, including raw states and states used to trigger interrupts and optionally wake.
- Configuration to indicate if this session will include some advanced features which can be turned off if done before device is active on bus, including HDR features.
- Data buffering, including simple buffering and a FIFO model, if enabled; but see below.

The Wrapper also permits external FIFO interface using a normal handshake to move 8 bits at a time in each direction. The tobus and frombus mechanisms still maintain a 1 or 2 byte holding buffer to isolate the FIFO from the clock domain crossing.

The Wrapper also has extended features like wake (raw and synchronized) which can be ignored.

The SlaveReset related signals are brought out, which are normally connected to the reset-detector block (not i2c SW reset).

Some status flags are also ported out, but they can be ignored.

The synchronous time-control port is only used if the external sync time control block is to be used.

4.3 Non-APB Full Wrapper details

The Non-APB Full wrapper exposes the same interfaces that the register block normally supports.

This means the wrapper is still handling the clock-domain-crossings via synchronization, but is relying on external nets to handle configurations, status, control of events (e.g. IBI), interrupts (if used), data (FIFOed or simple), and state info.

The clock-crossing interface maps many of the nets into **outp_** and **reg** and **outpflg_** and **regflg_** including combined nets. These are described below.

This allows for a more autonomous model in the system as well as handling of the registers by the system.

```
output       [18:8] outp_IntStates,   // interrupt status; see `IS_xx
input         [2:0] reg_EvPend,       // Event request. [2] is flag for change
input          [7:0] reg_EvIbiByte,    // optional byte (req if BCR said so)
output               outp_EvNoCancel,  // in IBI now, so no change to EvPend
output       [22:20] outp_EvDet,       // details of Event pending/done
output         [4:0] outp_GenErr,      // general errors from engines
output        [10:8] outp_DataErr,     // data errors from engines
input         [18:8] reg_clrIntStates, // clear int status when bits are 1
input          [4:0] reg_clrGenErr,    // clear errors when bits are 1
input         [10:8] reg_clrDataErr,   // clear errors when bits are 1
output               outpflg_MaxRd,    // Master changed MaxRd
output               outpflg_MaxWr,    // Master changed MaxWr
output        [11:0] outp_MaxRW,       // Value from Master for Rd or Wr
// now same CLK domain but related to data or FIFO
input                reg_TbEnd,        // end of to-bus data
input                reg_TbFlush,      // 1 cycle pulse to flush to-bus buffers
input                reg_FbFlush,      // 1 cycle pulse to flish from-bus holding
  // next 4 are only used if FIFO enabled
input          [5:4] reg_TxTrig,       // trigger level for TX (tb)
input          [7:6] reg_RxTrig,       // trigger level for RX (fb)
output       [20:16] outp_TxCnt,       // current counter for TX (tb)
output       [28:24] outp_RxCnt,       // current counter for RX (fb)
output               outp_TxFull,      // TX (tb) buffer or FIFO is full
output               outp_RxEmpty,     // RX (fb) buffer or FIFO is empty
// next set relate to internal FIFO (including ping-pong buffer)
  // next 3 will only be used to size(s) allowed by params
input          [1:0] regflg_wr_cnt,    // tb: 0=none, 1 = wrote byte
input          [7:0] reg_wdata,        // tb: byte
input          [1:0] regflg_rd_cnt,    // fb: 0=none, 1 = read byte
output         [1:0] outp_fb_ready,    // fb: 0=none available, 1,2,3 avail
output         [7:0] outp_fb_data,     // fb: byte
// Optional external FIFO:
  // To-bus means from slave to master
input                ixf_tb_avail,     // 1 if byte is available for tobus
input                ixf_tb_last,      // byte is last for message
input          [7:0] ixf_tb_data,      // actual data when avail=1
output               oxf_tb_start,     // pulsed when ixf_tb_avail==1 and START
output               oxf_tb_used,      // pulsed when tb_data used
  // From-bus means from master into slave
input                ixf_fb_free,      // 1 if space to take a byte frombus
output               oxf_fb_req,       // pulse when data to be taken (amd free=1)
output         [7:0] oxf_fb_data,      // data frombus when req=1
```

```
  output              oxf_fb_eof,       // frame ended in repeated START or STOP
```

4.4 Non-APB Mini Wrapper

Because some devices are very simple and want minimal functionality, the Mini Wrapper provides a raw interface straight off the SCL domain. It is up to the system to provide clock domain crossing synchronization where and if needed.

5. Pin/Pad controls for SCL and SDA

The peripheral uses 3 signals for each pad:

- pin_SCL_oena and pin_SDA_oena: This is active high.
    - Active high means that if this signal is 1, the pad should be driving output using the level/state in the corresponding _out pin (see below).
    - Active high means that if this signal is 0, the pad should be in High-Z.
    - Active high is used so that the pad is in Hi-Z except when intended to be driven, including when the peripheral is held in reset.
    - Note that pin_SCL_oena will be strapped 0 if HDR Ternary is not selected for.
- pin_SCL_out and pin_SDA_out: This is the output drive level of the pad when the corresponding _oena signal is 1. It has no effect when the _oena signal is 0.
    - A 1 means the pad should be pulling the line to Vddio (e.g. 1.8V or 3.3V).
    - A 0 means the pad should be pulling the line to GND (ie. 0V).
- pin_SCL_in and pin_SDA_in: This is the input from the pad.
    - It should be valid even if the corresponding _oena=1, but certainly when _oena=0.

Note the pin_SDA_oena_rise net is a tie-off (ignore) 1 bit output, unless using the External registered pad control, as explained in the next section.

5.1 External Registered Pad control

For systems where the I3C block would be far from the pads on the die or where the path timing is poor, the block provides an external registered pad control block.

The works by having the I3C block provide the combinatorial pre-states for SDA and pass across the die to the pad control. The pad control then uses the nearby SCL in-pad to drive its flops for SDA and so shortens the path from SCL to SDA to very little. This is shown below.

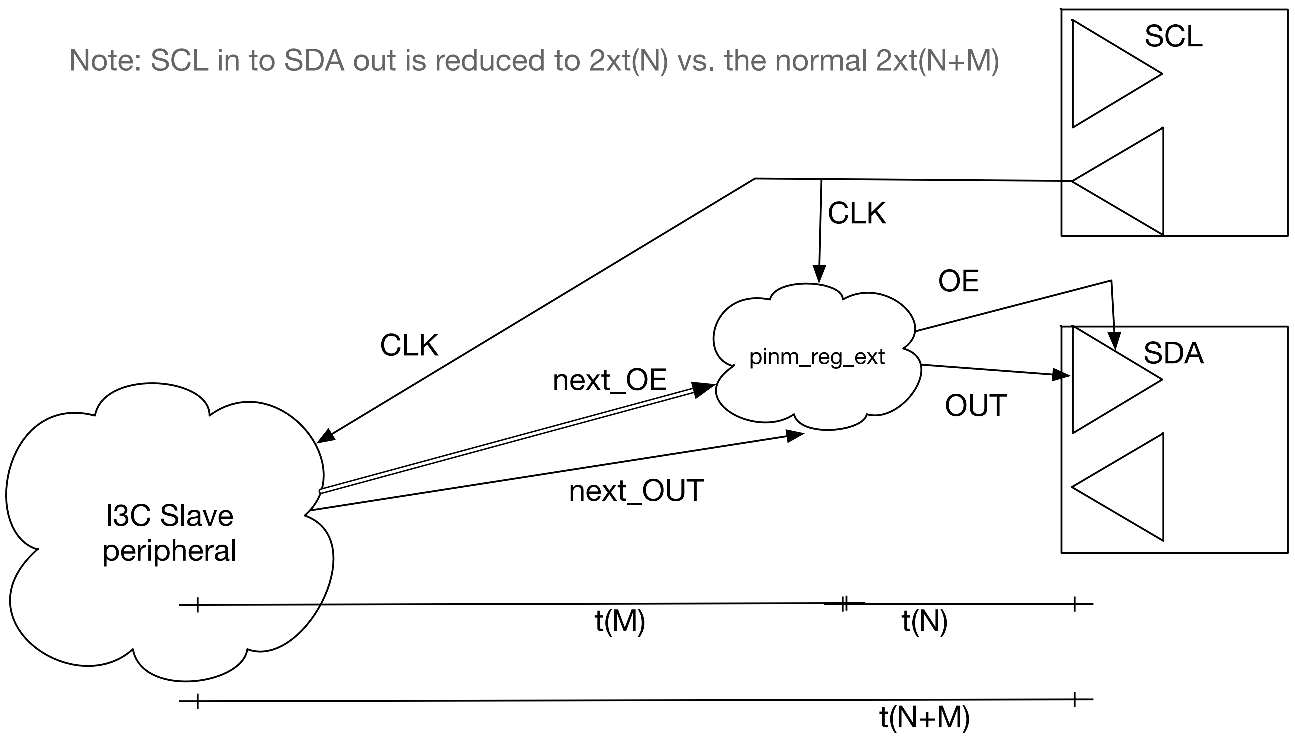


Figure 7 - showing the effect of using the external pad control block

5.2 Port interface to external pad control:

```
input                RSTn,              // master Reset – global or peripheral
// now the SCL as a pin, which we make into a clock and inverted clock. You can
// change falling edge to negedge logic if desired.
input                i_pad_SCL,         // direct from pad
// now the signals from the i3c slave peripheral
input                pin_SDA_out,       // out from peripheral
input                pin_SDA_oena,      // falling edge out-en from peripheral
// Connect rise0 below to rise[0] from peripheral, and 21 to rise[2:1]
input                pin_SDA_oena_rise0,// rising edge out-ena from peripheral
input                pin_SDA_oena_rise1,// force_SDA
input          [1:0] pin_SDA_oena_rise32,// rising edge out if HDR-DDR enabled, else 0
// the output to SDA pad
output               o_pad_SDA_oe,      // output enable for SDA
output               o_pad_SDA_out,     // out to SDA when oe=1
// now scan if used
input                scan_single_clock,// use scan_CLK uninverted if SCAN
input                scan_CLK
```

6. Autonomous Register access Slave interface

The autonomous register mechanism is described in the separate i3c_autonomous_slave document.