**Overview**

This document specifies I2C hardware IP functionality. This module conforms to the [Comportable guideline for peripheral functionality.](https://github.com/lowRISC/opentitan/blob/master/doc/contributing/hw/comportability/README.md) See that document for integration overview within the broader top level system.

**Features**

* Two-pin clock-data parallel bidirectional external interface: SCL (serial clock line) and SDA (serial data line)
* Support for I2C Controller ("I2C Master"1) and I2C Target ("I2C Slave"1) device modes
* Support for Standard-mode (100 kbaud), Fast-mode (400 kbaud) and Fast-mode Plus (1 Mbaud)
* Bandwidth up to 1 Mbaud
* Support for all "Mandatory" features as specified for I2C Controllers (as listed in Table 2 of the [I2C specification (revision 6)](https://web.archive.org/web/20210813122132/https:/www.nxp.com/docs/en/user-guide/UM10204.pdf)):
  + Start condition
  + Stop condition
  + Acknowledge (ACK)
  + 7-bit target address
* Support for the following optional capabilities:
  + Clock stretching in the controller mode
  + Automatic clock stretching in the target mode2
  + Programmable automatic ACK control in the target mode
* Support for multi-controller features:
  + Controller-controller clock synchronization
  + Controller bus arbitration
  + Target responses to broadcast transfers (similar to arbitration)
  + Bus idle detection
* Byte-formatted register interface with four separate queues: two queues in the controller mode, one for holding read data (RX), the other for holding bytes to be transmitted (TX: addresses or write data) and two queues in the target mode, for holding read (RX) and write (TX) data
* Direct SCL and SDA control in "Override mode" (for debugging)
* SCL and SDA ports mapped to I/O via the pinmux
* Interrupts in the controller mode for FMT and RX FIFO overflow, target NACK, SCL/SDA signal interference, timeout, unstable SDA signal levels, and transaction complete
* Interrupts in the target mode for TX FIFO empty during a read, TX FIFO nonempty at the end of a read, TX overflow and ACQ FIFO full, controller sending STOP after ACK, and controller ceasing to send SCL pulses during an ongoing transaction
* Loopback support with external controller when in target operation
* SW may reset I2C block using the Reset Manager

1 lowRISC is avoiding the fraught terms master/slave and defaulting to controller/target where applicable.

2 The target is only compatible with controllers that support clock stretching. For controllers that do not support clock stretching, it is expected that there must be an additional protocol to guarantee there is always sufficient space and data. These protocols are not in scope of this document.

**Description**

This IP block implements the [I2C specification (rev. 6)](https://web.archive.org/web/20210813122132/https:/www.nxp.com/docs/en/user-guide/UM10204.pdf), though with some variation in nomenclature. For the purposes of this document, an "I2C Controller" meets the specifications put forth for a "Master" device. Furthermore, a device which meets the specifications put forward for an "I2C Slave" device is here referred to as an "I2C Target" or "I2C Target Device". Note that there are some places where "host" is used, such as in register and interrupt names. This is equivalent to "controller" and comes from a time before more recent I2C and SMBus specifications settled on "controller" being the replacement. It should not be confused with "SMBus Host," which has a specific meaning in that protocol.

At a high-level, the I2C protocol is a clock-parallel serial protocol, with at least one controller issuing transactions to a number of targets on the same bus. A transaction may consist of multiple transfers, each of which has its own address byte and command bit (R/W). However, it is strongly recommended that all transfers in a transaction are issued to the same target address. Multiple controllers may begin a transaction simultaneously, but bus arbitration will cause only one stream of bits to reach a target. This IP block can detect and report lost bus arbitration, and typically, the next action for software is to retry the transaction.

Every transfer consists of a number of bytes transmitted, either from controller-to-target or target-to-controller. Each byte is typically followed by a single bit acknowledgement (ACK) from the receiving side. Typically each transfer consists of:

1. A START signal, issued by the controller.
2. An address, issued by the controller, encoded using 7 bits.
3. An R/W bit indicating if the transaction is a read-from or a write-to the target device. The R/W bit is encoded along with the address.
4. An acknowledge signal (ACK) sent by the target device.
5. Data bytes, where the number of bytes required is indicated by the controller, in a manner which differs between reads and writes.
   * For write transactions, the target device sends an ACK signal after every byte received. The controller indicates the end of a transaction by sending a STOP or repeating the START signal.
   * For read transactions, the target device continues to send data as long as the controller acknowledges the target-issued data by sending an ACK signal. Once the controller has received all the required data it indicates that no more data is needed by explicitly de-asserting the ACK signal (this is called a NACK signal) just before sending a STOP or a repeated START signal.
6. A STOP signal or a repeated START signal.

All well-behaved transactions end with a STOP signal, which indicates the end of the last transfer and when the bus is free again. However, this IP does support detection of bus timeouts, when there has been no activity for a specified amount of time.

This protocol is generally quite flexible with respect to timing constraints, and slow enough to be managed by a software microcontroller, however such an implementation requires frequent activity on the part of the microcontroller. This IP presents a simple register interface and state-machine to manage the corresponding I/O pins directly using a byte-formatted programming model.

**Compatibility**

This IP block should be compatible with any target device covered by the [I2C specification (rev. 6)](https://web.archive.org/web/20210813122132/https:/www.nxp.com/docs/en/user-guide/UM10204.pdf), operating at speeds up to 1 Mbaud. This IP in the controller mode issues addresses in 7-bit encoding, and in the target mode, receives addresses in 7-bit encoding. (It remains the obligation of system designers to ensure that devices remain in a 7-bit address space.) This IP also supports clock-stretching, should that be required by target devices.

**Theory of Operation**

**Functional Description**

I2C IP is a controller-target combo that can function as an I2C controller and/or an I2C target. These functional modules are enabled at runtime by setting the register fields [CTRL.ENABLEHOST](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#ctrl) and [CTRL.ENABLETARGET](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#ctrl). If the Controller Module is to be used in a multi-controller environment, [CTRL.MULTI\_CONTROLLER\_MONITOR\_EN](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#ctrl) would also need to be set. Note the ordering requirements in the register description for multi-controller configurations.

The SCL and SDA outputs from the block combine the SCL and SDA outputs from the Controller Module and Target Module. The modules operate as though they were independent devices on the same I2C bus, with any logic low value taking priority.

On the input path, the IP's clock samples SCL and SDA after a 2-flop synchronizer. However, the IP's input pins do not reflect the start point for the full input path delay, since pads and nets from I/O buffers also contribute. The RTL parameter InputDelayCycles must be set to the maximum number of cycles of delay from the IP's output pin driving SCL or SDA to the IP's corresponding input pin sensing the transition. InputDelayCycles can exclude the rise time that would be captured by [TIMING1.T\_R](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timing1). The reference point for InputDelayCycles is the clock edge after the launch clock edge. In other words, a value of 0 means that the IP's 2-flop synchronizer would sample the IP's own outputs within one clock cycle (if the external pad driver could switch instantaneously).

**Bus Monitor**

The Bus Monitor block examines the SCL and SDA inputs and any transmitted outputs from the Controller Module and Target Module. It detects START and STOP conditions and forwards the events. In addition, it tracks whether a bus transaction is currently active and informs the Controller Module when the bus is free for it to transmit a new transaction. Finally, it detects when bus arbitration is lost and informs the modules when they must release the bus and allow other devices to communicate.

For signaling when the bus is free, the behavior depends on whether [CTRL.MULTI\_CONTROLLER\_MONITOR\_EN](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#ctrl) is enabled. If it's not enabled, the monitor merely observes the bus free time betwen transactions.

If multi-controller monitor mode is enabled, then the monitor also tracks transactions from other controllers, and it will report that the bus is busy until the free time expires after an external controller issues a STOP condition. The bus can also become free after the [HOST\_TIMEOUT\_CTRL](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#host_timeout_ctrl) duration expires, if the bus was left in the idle state, or it can become free in the case of a bus timeout (if enabled in [TIMEOUT\_CTRL](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timeout_ctrl)).

At initial attachment in a multi-controller environment, typically the controller needs to wait until it determines the bus is free, and the initial state should be assumed to be busy. To make this happen, [HOST\_TIMEOUT\_CTRL](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#host_timeout_ctrl) should be programmed before enabling the bus monitor in [CTRL](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#ctrl), and [CTRL.MULTI\_CONTROLLER\_MONITOR\_EN](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#ctrl) should be set either before or at the same time as enabling the Controller Module with [CTRL.ENABLEHOST](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#ctrl).

**Controller Module**

The Controller Module implements all mandatory controller features of the I2C protocol for multi-controller configurations, including clock synchronization and bus arbitration. It also implements support for handling clock stretching.

The state-machine-controlled Controller Module allows for higher-speed operation with less frequent software interaction. In this module, the I2C pins are controlled by the state machine, which in turn is controlled by a sequence of formatting indicators. These indicators determine:

* The sequence of bytes which should be transmitted on the SDA and SCL pins.
* The periods between transmitted bytes when the state-machine should stop transmission and instead read back a fixed number of bytes.
* Which bytes should be preceded by a START symbol.
* Which bytes should be followed by a STOP symbol.

The format indicator consists of 13-bits. That is of one single format byte (entered into the format FIFO through [FDATA.FBYTE](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#fdata)), and five 1-bit flags (entered into the format FIFO through registers [FDATA.READB](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#fdata), [FDATA.RCONT](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#fdata), [FDATA.START](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#fdata), [FDATA.STOP](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#fdata) and [FDATA.NAKOK](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#fdata))

The I2C Control Module reads each format indicator from the head of FMT\_FIFO and processes them in turn. If none of the flags are set for the format indicator, the I2C FSM simply transmits the format byte onto the SCL and SDA pins according to the specification, waits for acknowledgement, and then proceeds to the next format indicator. The format flags modulate the behavior as follows:

* READB (corresponds to [FDATA.READB](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#fdata)): Signifies the format byte ([FDATA.FBYTE](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#fdata)) should be treated as an unsigned number, R, and prompts the state machine to read R bytes from the target device. Bytes read from the bus are inserted into the RX FIFO where they can be accessed by software. A value of 0 is treated as a read of 256 bytes. To read a larger byte stream, multiple 256-byte reads can be chained together using the RCONT flag.
* RCONT (corresponds to FIFO inputs [FDATA.RCONT](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#fdata), only used with READB):
  + If RCONT is set, the format byte represents part of a longer sequence of reads, allowing for reads to be chained indefinitely.
  + The RCONT flag indicates the final byte returned with the current read should be responded to with an ACK, allowing the target to continue sending data. (Note that the first R-1 bytes read will still be acknowledged regardless of whether RCONT is asserted or not.)
* START (corresponds to [FDATA.START](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#fdata), Ignored when used with READB): Issue a START condition before transmitting the format byte on the bus. This flag may also be used to issue a repeated start condition.
* STOP (corresponds to [FDATA.STOP](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#fdata)): Issue a STOP signal after processing this current entry in the FMT FIFO. Note that this flag is not compatible with (READB & RCONT), and will cause bus conflicts.
* NAKOK (corresponds to [FDATA.NAKOK](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#fdata), Not compatible with READB): Typically every byte transmitted must also receive an ACK signal, and the IP will raise an exception if no ACK is received. However, there are some I2C commands which do not require an ACK. In those cases this flag should be asserted with FBYTE indicating no ACK is expected and no interrupt should be raised if the ACK is not received.

The Controller Module may proceed through all commands in the FMT FIFO as long as no exceptional event occurs. If the Controller Module receives an unexpected NACK, a bus timeout occurs, or arbitration is lost while the Controller Module is transmitting, the module halts. The module will not continue processing commands without software intervention.

Note that a full RX FIFO does not stop command processing. Ensure the RX FIFO has enough space, else overflows can occur.

**Controller NACK Handling**

If the Controller Module transmits a byte and the 9th bit is a NACK from the Target/Bus, the controller\_halt interrupt is usually asserted (modulo the effect of [FDATA.NAKOK](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#fdata)). If the controller\_halt interrupt is asserted, the Controller FSM will halt until the interrupt condition has been cleared in the [CONTROLLER\_EVENTS.NACK](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#controller_events) register. Software will likely want to do one of two things in handling this irq:

1. End the current transaction by setting [CTRL.ENABLEHOST](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#ctrl) to 1'b0, and clear the FIFOs to be ready to begin a new transaction.
2. Clear the FIFOs, then begin a new transfer without ending the transaction by adding new FDATA indicators to the FMTFIFO. In this case, the first entry written to the FMTFIFO should have [FDATA.START](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#fdata) set to 1'b1 to create a repeated start condition.

Software should then clear the [CONTROLLER\_EVENTS.NACK](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#controller_events) to proceed with the next operation. The controller\_halt interrupt is a CIP status type, so it doesn't clear by writing to the [INTR\_STATE.CONTROLLER\_HALT](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#intr_state) bit. Instead, the CONTROLLER\_EVENTS CSR tracks the contributing events, and all must be cleared for the interrupt to de-assert.

The Halt-on-NACK behaviour may be problematic if SW is not responsive, so there is a timeout mechanism that can automatically end the transaction by creating a STOP condition and returning to halt in an idle state. This is configured using the [HOST\_NACK\_HANDLER\_TIMEOUT](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#host_nack_handler_timeout) CSR, which allows software to configure the delay before hardware will terminate the transaction. When the timeout occurs, the [CONTROLLER\_EVENTS.UNHANDLED\_NACK\_TIMEOUT] bit will set, and this condition will also cause the controller\_halt interrupt to assert (or remain asserted). Like the original NACK event, the timeout bit will cause the Controller Module to remain halted until it is cleared. To continue with a new transaction, software should ensure that the controller\_halt interrupt has been handled, with all contributing event bits cleared.

When the FSM first halts due to an unexpected NACK, there is still an ongoing transaction. The Controller Module stops immediately following the (N)ACK bit's hold time after the SCL falling edge, and SCL remains asserted low. If a NACK handling timeout occurs or [CTRL.ENABLEHOST] is cleared, then the FSM will halt again after the STOP condition is sent. The STOP condition terminates the transaction, leaving the FSM halted in the "bus idle" state, with SDA and SCL released.

**Bus Arbitration**

In a multi-controller configuration, the Controller Module may begin transmitting at the same time as another controller. The Bus Monitor will detect when the Controller Module loses arbitration, such as when it attempts to transmit a logic high, but another device is pulling SDA low. The monitor will inform the Controller Module, and the Controller Module will abort its command and halt. The [CONTROLLER\_EVENTS.ARBITRATION\_LOST](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#controller_events) bit will be set, and the controller\_halt interrupt state will reflect that the module is halted.

From here, software must decide what to do. The FMT FIFO will have the entries that remained, but typically, this event will result in software retrying the full transaction. In that case, software would clear the FMT FIFO (and possibly the RX FIFO), then load the FMT FIFO with the original commands, starting from the beginning of the transaction. All bits in [CONTROLLER\_EVENTS](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#controller_events) must be cleared for the Controller Module to release from the halted state.

**Target Module**

The Target Module implements all mandatory features for targets in multi-controller configurations, as enumerated in the I2C specification. It also supports clock stretching from that same specification.

In addition, the Target Module has the necessary mechanisms to support mandatory SMBus 3.0 features, as well as dynamically-assigned addresses using the SMBus Address Resolution Protocol. Address bytes may be configured to be always ACK'd, and the Target ACK Control feature may be used to stretch the clock until software can decide whether to ACK the incoming byte. The Target Module can also be configured to observe SMBus's maximum cumulative target clock extension time. There is no hardware support for calculating or checking the Packet Error Code, however; these values would need to be handled in software.

**Target Address Registers**

Every transfer involving the I2C Target Module begins with an address. The I2C Target Module supports up to two 7-bit address and 7-bit mask pairs. The Target Module accepts a transaction if the result of the bitwise AND operation performed on the transaction address sent by the controller and a mask matches the assigned address corresponding to the mask. In other words, address matching is performed only for bits where the mask is "1". Thus, with the masks set to all ones (0x7F), the target device will respond to either of the two assigned unique addresses and no other. If the mask and the assigned address both have zeros in a particular bit position, that bit will be a match regardless of the value of that bit received from the controller. Note that if, in any bit position, the mask has zero and the assigned address has one, no transaction can match and such mask/address pair is effectively disabled.

Finally, in the special case where the mask is all zeroes (0x00), then that corresponding entry is ignored and will not match. If the user wishes to have the target respond to *all* addresses, then this can be achieved by setting mask0 and mask1 to 0x01 and setting address0 to 0x00 and address1 to 0x01.

The assigned address and mask pairs are set in registers [TARGET\_ID.ADDRESS0](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#target_id), [TARGET\_ID.MASK0](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#target_id), [TARGET\_ID.ADDRESS1](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#target_id), and [TARGET\_ID.MASK1](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#target_id).

**Target Transfers**

When the Target Module accepts a transfer, it acquires the transfer address, read/write bit, and START signal sent by the controller and inserts them into ACQ FIFO, where they can be accessed by software. [ACQDATA](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#acqdata) represents the ACQ FIFO output. See the register description for more information about the representation.

If the transfer is a write operation (R/W bit = 0), the target proceeds to read bytes from the bus and insert them into ACQ FIFO until the controller terminates the transfer by sending a STOP or a repeated START signal. A STOP or repeated START indicator is inserted into ACQ FIFO as the next entry following the last byte received, in which case other bits may be junk or the target address, respectively.

Note that the repeated START indicator only appears if it matches the address. Typically, this is the case, and a transaction that changes targets between transfers is strongly discouraged. If no additional transfers targeting this target's address occur, an entry representing the end of the transfer will wait for the STOP symbol.

If the transaction is a read operation (R/W bit = 1), the target pulls bytes out of TX FIFO and transmits them to the bus until the controller signals the end of the transfer by sending a NACK signal. Under certain conditions, the target will hold SCL low to stretch the clock and give software time to write data bytes into TX FIFO or handle the available command. The conditions include the following:

* TX FIFO holding no data
* the ACQ FIFO containing more than the 1 entry representing the address/command byte
* [TARGET\_EVENTS](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#target_events) holding at least one unhandled event

See [stretching during read](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/theory_of_operation.md#stretching-during-read) for more details. Note that data for TX FIFO is pushed through the register [TXDATA](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#txdata).

Typically, a NACK signal is followed by a STOP or repeated START signal; the IP will raise an exception if the controller sends a STOP signal after an ACK. The ACK/NACK signal is represented in the MSB of [ACQDATA.SIGNAL](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#acqdata), in the same entry as the STOP or repeated START signal. For ACK signals, the value of the MSB is 0; for NACK signals, the value is 1.

**Target Clock Stretching**

As described in the I2C specification, a target device can pause a transaction by holding SCL low. There are 3 cases in which this design stretches the clock:

* After the target receives the address but before pushing it to the ACQ FIFO
* Before the (N)ACK bit completes transmission during a write transfer
* After the ACK bit is sent by the controller-transmitter during a read transfer

**Stretching after address read**

When a target device receives a start, the address and R/W bit are written into the ACQ FIFO. If there is no space in the ACQ FIFO to receive such a write, the target stretches the clock after the ACK bit and waits for software to make space. The acq\_stretch interrupt is generated to alert software to such a situation.

**Stretching during write**

If the controller tries to write a data byte into the ACQ FIFO when there is no available space, the clock is stretched before the rising edge of SCL on the ACK bit. In addition, the Target Module may stretch if Target ACK Control Mode is enabled and the Auto ACK Counter has reached 0. See the [Target ACK Control Mode](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/theory_of_operation.md#target-ack-control-mode) section for more about it.

The acq\_stretch interrupt is generated to alert software to these situations.

**Stretching during read**

When a target device receives a start and read command, it may stretch the clock for any of the following reasons:

* If there is no data available to be sent back (TX FIFO empty case), the target stretches the clock until data is made available by software.
* If there is more than 1 entry in the ACQ FIFO.
  + Having more than 1 entry in the ACQ FIFO suggests there is potentially an unhandled condition (STOP / RESTART) or an unhandled command (START) that requires software intervention before the read can proceed.
* If there is an unhandled event in [TARGET\_EVENTS](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#target_events).

The tx\_stretch interrupt is generated to alert software to such a situation.

**Target NACKs**

When the Target Module replies with a NACK during the (N)ACK phase, it indicates that it is unable to accept the byte or command. The NACK response is sticky until the end of the transaction. In other words, once the Target Module issues its first NACK, it will NACK all subsequent bytes in the transaction, with one caveat.

Assuming [CTRL.NACK\_ADDR\_AFTER\_TIMEOUT](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#ctrl--nack_addr_after_timeout) is 0, a matching address will always elicit an ACK response, which is required by SMBus. However, if the Target Module had already issued a NACK in this transaction, that byte will not be pushed to the ACQ FIFO. After committing to NACKing the transaction, the Target Module will leave SDA released high for all bits except the ACK of a matching address byte.

The decision to NACK will show up in the ACQ FIFO by the end of the first transaction to be NACK'd. The final space in the ACQ FIFO is reserved for the STOP condition of a transaction, so in all cases where a START signal reached the ACQ FIFO, that first NACK'd transaction will always end with a NACK\_STOP signal. For write transactions, the particular NACK'd data byte will also be in the ACQ FIFO if there was space.

If software does not handle any persisting issues that caused NACKing and the next transaction leads to the Target Module NACKing again, all associated data will be dropped while the ACQ FIFO is full.

The decision to NACK the transaction can occur for any of the following reasons:

* A timeout after stretching the clock
  + If enabled by [TARGET\_TIMEOUT\_CTRL](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#target_timeout_ctrl)
* Bus arbitration for the target was lost, such as during SMBus ARP broadcast commands
* The controller's SCL low period is detected to be too short for a response
* Software directed the Target Module to NACK
  + Software has some control if ACK Control Mode is enabled

**Target ACK Control Mode**

ACK Control Mode is a feature that allows software to take control of ACK/NACK decisions during a Write transfer. When ACK Control Mode is disabled in [CTRL.ACK\_CTRL\_EN](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#ctrl), the Target module will automatically ACK any incoming data byte, so long as there is sufficient space in the ACQ FIFO. If there isn't enough space for the data byte and a STOP, the Target module will stretch the clock until space is made available (or a stretch timeout occurs). However, if ACK Control Mode is enabled, the [TARGET\_ACK\_CTRL](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#target_ack_ctrl) CSR can also cause the Target module to stretch the clock at the (N)ACK phase.

In ACK Control Mode, an Auto ACK Counter influences whether the Target module automatically ACKs a data byte when there is space in the ACQ FIFO. The counter begins at 0 for every transfer, and it decrements for every data byte that is ACK'd. In addition, this counter only applies to the bytes in a Write transfer that follow the address byte. When a transfer reaches the (N)ACK phase and the [TARGET\_ACK\_CTRL.NBYTES](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#target_ack_ctrl) CSR is 0, the Target module stretches the clock and awaits instructions from software, up to the stretch timeout. Note that because the Auto ACK Counter begins every transfer at 0, the first data byte will always stretch the clock.

To release SCL before the timeout, software can choose whether to ACK or NACK the byte. To ACK the byte, software would program the Auto ACK Counter to a greater number, and if the value programmed is greater than 1, it would automatically ACK subsequent bytes as well. Even with a nonzero count, though, the Target module could still stop and stretch the clock if the ACQ FIFO doesn't have sufficient space. Otherwise, software can explicitly NACK that byte and the rest of the transaction by writing a 1 to [TARGET\_ACK\_CTRL.NACK](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#target_ack_ctrl). If the Target module is not stretching the clock, writes to [TARGET\_ACK\_CTRL](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#target_ack_ctrl) are ignored.

With the data byte pending in the (N)ACK phase, it hasn't been written to the ACQ FIFO yet. Instead, software can read the [ACQ\_FIFO\_NEXT\_DATA](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#acq_fifo_next_data) CSR to decide what to do. Note that after software indicates its decision, this pending byte will still enter the ACQ FIFO if there is space, accompanied by the metadata reflecting whether it was ACK'd or NACK'd.

This feature is intended to support protocols that require mid-transfer NACK decisions based on the current data transferred (e.g. as in SMBus).

**Timing Control Registers**

For Standard-mode, Fast-mode and Fast-mode Plus, the timing requirements for each transaction are detailed in Table 10 of the [I2C specification (rev. 6)](https://web.archive.org/web/20210813122132/https:/www.nxp.com/docs/en/user-guide/UM10204.pdf). To claim complete compatibility at each mode, the state machine timings need to be adapted to whether there are Standard-mode, Fast-mode and Fast-mode Plus targets on the bus. Furthermore, depending on the actual capacitance of the bus, even a bus with all Fast-mode Plus capable targets may have to operate at slower speeds than 1 Mbaud. For example, the controller may need to run at lower frequencies, as discussed in Section 5.2 of the specification, but the computation of the nominal frequency will depend on timing specifications in Table 10, in this case particularly, the limits on tLOW, tHIGH, tr, and tf. Assuming no clock stretching, for a given set of these four parameters the baud rate is then given to be: $$ 1/f\_{SCL}=t\_{LOW}+t\_{HIGH}+t\_{r}+t\_{f}. $$

Thus in order to ensure compliance with the spec in any particular configuration, software will program the I2C controller IP with explicit values for each of the following timing parameters, as defined in Figure 38 of the specification.

* tLOW: set in register [TIMING0.TLOW](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timing0).
* tHIGH: set in register [TIMING0.THIGH](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timing0).
* tr: set in register [TIMING1.T\_R](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timing1). (Note: The rise time cannot be explicitly controlled by internal hardware, and will be a function of the capacitance of the bus. Thus this parameter is largely budgetary, meaning that it tells the state machine how much time to wait for an RC rise.)
* tf: set in register [TIMING1.T\_F](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timing1). (Note: The fall time cannot be explicitly controlled by internal hardware, and is a function of the pin driver. Thus this parameter is also budgetary. Given that the actual fall time cannot be controlled to stay above the minimum values set in Table 10 of the specification, and so this in this regard this module currently is not strictly compliant to the I2C spec. The system design is responsible for meeting the spec's minimum fall time. )
* tSU,STA: set in register [TIMING2.TSU\_STA](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timing2)
* tHD,STA: set in register [TIMING2.THD\_STA](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timing2)
* tSU,DAT: set in register [TIMING3.TSU\_DAT](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timing3). Taken to be synonymous with TSU,ACK
* tHD,DAT: set in register [TIMING3.THD\_DAT](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timing3). Taken to be synonymous with THD,ACK. Moreover, since the pin driver fall time is likely to be less then one clock cycle, this parameter is also taken to be synonymous with the parameters TVD,DAT and TVD,ACK This parameter controls the number of cycles after the falling edge of SCL that SDA is driven. In addition, when the IP operates as a target, the parameter specifies the required SCL high time after SDA changes to satisfy the Start and Stop symbol decoders.
* tSU,STO: set in register [TIMING4.TSU\_STO](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timing4).
* tBUF: set in register [TIMING4.T\_BUF](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timing4)

The values programmed into the registers [TIMING0](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timing0) through [TIMING4](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timing4) are to be expressed in units of the input clock period. It is important that the internal clock is at least 50x the bus clock so that the proportions of the timings can be accurately captured. Note in order to ensure compliance with the I2C spec, firmware must program these registers with values within the ranges laid out in Table 10 of the specification. These values can be directly computed using DIFs given the desired speed standard, the desired operating frequency, and the actual line capacitance. These timing parameters are then fed directly to the I2C state machine to control the bus timing.

A detailed description of the algorithm for determining these parameters--as well as a couple of concrete examples--are given in the [Programmer's Guide](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/programmers_guide.md#timing-parameter-tuning-algorithm).

Note that the IP also has minimum values for [TIMING0.TLOW](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timing0), [TIMING0.THIGH](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timing0), [TIMING2.THD\_STA](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timing2), and [TIMING3.THD\_DAT](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timing3). THD\_DAT must be at least 1, and to disambiguate START conditions from early SDA arrival, THD\_STA must be at least 2 cycles longer than THD\_DAT.

The low count must be long enough to cover the round-trip time of the Controller Module's SCL drive to the target through the sampling of the target's data driven on SDA. As there is an output flop, a 2-flop synchronizer, a rise time, and a pad input delay all on the path, TLOW must be at least 3 cycles + T\_R + InputDelayCycles.

Similarly, the high count must be long enough to cover the a round-trip time to sense outputs for various conditions (e.g. lost arbitration, target clock stretching, etc.). THIGH must be at least 4 cycles + InputDelayCycles.

**Timeout Controls**

**Bus Timeout or Stretch Timeout**

A malfunctioning (or otherwise very slow) target device can hold SCL low indefinitely, stalling the bus. In addition, in an SMBus context, a controller can purposefully hold SCL low for an extended period of time (tTIMEOUT:MIN) to force a bus timeout. For these reasons, [TIMEOUT\_CTRL](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timeout_ctrl) provides a timeout mechanism for determining one of two variants of an SCL low timeout.

Enabling a timeout at all requires setting [TIMEOUT\_CTRL.EN](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timeout_ctrl) and configuring [TIMEOUT\_CTRL.VAL](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timeout_ctrl) with the desired number of cycles to count before timing out. With the configuration of [TIMEOUT\_CTRL.MODE](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timeout_ctrl), one of two types of timeouts can be selected.

If a STRETCH\_TIMEOUT is selected, the Controller Module detects when some other device holds SCL low for more cycles than than [TIMEOUT\_CTRL.VAL](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timeout_ctrl) clock ticks. If the counter expires, the stretch\_timeout interrupt will be asserted.

If a BUS\_TIMEOUT is selected, the SCL low time accumulates across all sources, including this IP's Controller Module. This counter mode can be used to detect SMBus's definition of a bus timeout. If a bus timeout occurs while the Controller Module is active, the corresponding [CONTROLLER\_EVENTS.BUS\_TIMEOUT](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#controller_events) bit is set, halting the Controller Module and requiring handling from software before the next command will process.

Similarly, if a bus timeout occurs while the Target Module was addressed in the current transaction, a NACK\_STOP signal is added to the ACQ FIFO. In addition, if the last command byte addressing the module issued a read operation, [TARGET\_EVENTS.BUS\_TIMEOUT](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#target_events) will be set, causing the Target Module to stretch the clock on read operations until software handles the event.

For both cases, the counter resets when SCL goes high, so this count only accumulates during a single bit transfer.

**SCL Idle Timeout**

The I2C IP provisions a counter for detecting when SCL has remained high for too long during an active transaction. [HOST\_TIMEOUT\_CTRL](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#host_timeout_ctrl) is optionally enabled, determined by whether the register value is nonzero.

If this timeout occurs in multi-controller monitor mode with SDA released high, the Bus Monitor uses this as a "bus idle" condition, which allows the Controller Module to begin transmitting. This timeout can be used to implement SMBus's tHIGH,MAX bus idle condition. In addition, the SCL Idle Timeout is used by the Bus Monitor to set up waiting for a bus idle condition at init, if multi-controller monitor mode was enabled no later than the Controller Module.

If an active transfer is targeting this IP's Target Module, this timeout will trigger a host\_timeout interrupt.

Unlike the bus timeout, the idle timeout does not trigger an end to the command, so the controller could potentially recover. Instead, the interrupt is merely informative. However, a START condition from any device will terminate the transfer, and that includes the Controller Module. Note that this behavior could cause transaction boundaries to be lost; the Target Module will report that this next START condition is a repeated START.

**Target Module Stretch Timeout**

If enabled with [TARGET\_TIMEOUT\_CTRL.EN](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#target_timeout_ctrl), the Target Module will automatically NACK transactions if it stretches the clock for more cycles than [TARGET\_TIMEOUT\_CTRL.VAL](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#target_timeout_ctrl). This counter accumulates across an entire transaction, and with the correct choice of timeout value, it complies with SMBus's specified maximum cumulative clock extension time for the target.

**Interrupts**

The I2C module has a few interrupts including general data flow interrupts and unexpected event interrupts.

**Controller Module**

Whenever the RX FIFO exceeds the designated number of entries, the interrupt rx\_threshold is asserted to inform firmware. Firmware can configure the threshold value via the register [HOST\_FIFO\_CONFIG.RX\_THRESH](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#host_fifo_config).

Whilst the FMT FIFO level lies below a designated number of entries, the fmt\_threshold interrupt is asserted. Firmware can configure the threshold value via the register [HOST\_FIFO\_CONFIG.FMT\_THRESH](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#host_fifo_config).

If the RX FIFO receives an additional write request when its FIFO is full, the interrupt rx\_overflow is asserted and the character is dropped.

If the module transmits a byte and the 9th bit is a NACK from the Target/Bus, the nack interrupt is usually asserted (modulo the effect of [FDATA.NAKOK](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#fdata)). If the 'nack' interrupt is asserted, the Controller Module FSM will halt until the interrupt has been acknowledged. See [the Controller NACK handling section](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/theory_of_operation.md#controller-nack-handling) above for more details on this behaviour.

When the I2C module is in transmit mode, the scl\_interference or sda\_interference interrupts will be asserted if the IP identifies that some other device (controller or target) on the bus is forcing either signal low and interfering with the transmission. It should be noted that the scl\_interference interrupt is not raised in the case when the target device is stretching the clock. (However, it may be raised if the target allows SCL to go high and then pulls SCL down before the end of the current clock cycle.)

A target device should never assert 0 on the SDA lines, and in the absence of multi-controller support, the sda\_interference interrupt is raised whenever the controller IP detects that another device is pulling SDA low.

On the other hand, it is legal for the a target device to assert SCL low for clock stretching purposes. With clock stretching, the target can delay the start of the following SCL pulse by holding SCL low between clock pulses. However the target device must assert SCL low before the start of the SCL pulse. If SCL is pulled low during an SCL pulse which has already started, this interruption of the SCL pulse will be registered as an exception by the I2C core, which will then assert the scl\_interference interrupt.

Though normal clock stretching does not count as SCL interference, if the module detects that a target device has held SCL low and stretched the any given SCL cycle for more than [TIMEOUT\_CTRL.VAL](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timeout_ctrl) clock ticks this will cause the stretch timeout interrupt to be asserted. This interrupt is suppressed, however, if [TIMEOUT\_CTRL.EN](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#timeout_ctrl) is deasserted low.

Except for START and STOP symbols, the I2C specification requires that the SDA signal remains constant whenever SCL is high. The sda\_unstable interrupt is asserted if, when receiving data or acknowledgement pulse, the value of the SDA signal does not remain constant over the duration of the SCL pulse.

Transactions are terminated by a STOP signal. The controller may send a repeated START signal instead of a STOP, which also terminates the preceding transaction. In both cases, the cmd\_complete interrupt is asserted, in the beginning of a repeated START or at the end of a STOP.

**Target Module**

The interrupt cmd\_complete is asserted whenever a RESTART or a STOP bit is observed by the target.

The interrupt tx\_stretch is asserted whenever target intends to transmit data but cannot. See [stretching during read]({{< relref "#stretching-during-read" >}}).

When a controller receives enough data from a target, it usually signals the end of the transaction by sending a NACK followed by a STOP or a repeated START. In a case when a target receives a STOP without the prerequisite NACK, the interrupt unexp\_stop is asserted. This interrupt just means that a STOP was unexpectedly observed during a controller read. It is not necessarily harmful, but software can be made aware just in case.

If the ACQ FIFO exceeds the designated number of entries, the interrupt acq\_threshold is raised to inform firmware. Firmware can configure the threshold value via the register [TARGET\_FIFO\_CONFIG.ACQ\_THRESH](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#target_fifo_config).

Whilst the TX FIFO level is below a designated number of entries the tx\_threshold interrupt is asserted. Firmware can configure the threshold value via the register [TARGET\_FIFO\_CONFIG.TX\_THRESH](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#target_fifo_config).

If firmware sets the bit [TARGET\_FIFO\_CONFIG.TXRST\_ON\_COND](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#target_fifo_config), the TX FIFO will be reset whenever a RSTART or STOP condition is seen on the bus during an active Target-Mode transaction. This behaviour may be useful to software, as any remaining data in the TXFIFO after a Sr/P condition is probably no-longer applicable to the next transfer, so will likely have to be cleared out anyway. Keeping this behaviour as a toggle allows software to observe the fifo state before resetting it, which may be useful to understand how much of the previous transfer completed if that information is helpful or relevant.

If the Target module stretches the clock as a target-receiver, the interrupt acq\_stretch is asserted. This can be due to a full ACQ FIFO, or it can be due to an ACK Control Mode stretch request. If ACK Control Mode is enabled, check the relevant bits in [STATUS](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#status) to determine the reason(s). The acq\_stretch interrupt is a Status type, so it will only de-assert once the stretch conditions are cleared.

If a controller ceases to send SCL pulses at any point during an ongoing transaction, the target waits for a specified time period and then asserts the interrupt host\_timeout. Controller sending an address and R/W bit to all target devices, writing to the selected target, or reading from the target are examples of ongoing transactions. The time period is counted from the last low-to-high SCL transition. Firmware can configure the timeout value via the register [HOST\_TIMEOUT\_CTRL](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#host_timeout_ctrl).

**Implementation Details: Format Flag Parsing**

To illustrate the behavior induced by various flags added to the formatting queue, the following figure shows a simplified version of the I2C\_Controller state machine. In this simplified view, many sequential states have been collapsed into four sub-sequences of states (shown in square brackets) or have their names abbreviated:

* Issue start
* Issue stop
* Transmit Byte
* Read Bytes

Within each of these sub-sequences, state transitions depend only on the SDA/SCL inputs or internal timers. Each sub-sequence has a terminal event--generically labeled "[completed]" which prompts the transition to another sequence or state.

**Application Notes**

**Virtual Open Drain**

In devices which lack a true open drain buffer functionality, this IP implements a "virtual Open Drain" functionality. The SDA and SCL outputs are assumed to be connected to a tri-state buffer, with independent enable outputs for both signals.

Rather than toggling the buffer inputs, the buffer inputs are *continuously asserted low*, and instead the buffer *enable* signals are toggled. The SDA or SCL buffers are enabled for a logical "Low" output on the respective signal, and are disabled for logical "High" outputs. This arrangement allows the output pins to float high if there is no conflict from external devices, or to be pulled low if there is a conflict (as is required for clock-stretching or multi-controller functionality).

This arrangement is necessary for FPGA builds.

**Glitch Filter**

The IP does not currently implement the specification's required 50 ns glitch filter. For full spec compliance, the glitch filter must be added to the system design.

**Override Mode for Direct Pin Access**

The I2C hardware interface consists of two external pins, SCL and SDA, whose behavior is described in the [I2C specification (rev. 6)](https://web.archive.org/web/20210813122132/https:/www.nxp.com/docs/en/user-guide/UM10204.pdf). These pins are typically controlled by an internal state machine. However, there is a simpler "override" mode, by which these pins can be directly manipulated by software. This override mode is useful for both troubleshooting and error recovery.

To enter override mode, the register field [OVRD.TXOVRDEN](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#ovrd) is asserted by software. In this state the output drivers scl\_tx\_o and sda\_tx\_o are controlled directly by the register fields [OVRD.SCLVAL](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#ovrd) and [OVRD.SDAVAL](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#ovrd). When [OVRD.SCLVAL](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#ovrd) and [OVRD.SDAVAL](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#ovrd) are set high, the virtual open drain configuration will leave the output resistively pulled high, and controllable by remote targets. In this state, with SCL or SDA asserted high, the register fields [VAL.SCL\_RX](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#val) and [VAL.SDA\_RX](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/registers.md#val) can be used to receive inputs (including remote acknowledgments) from target devices.

**FSM control of SCL and SDA**

SCL and SDA are generated through the internal state machine of the controller and target modules. Since SCL is directly decoded from the states, it can have short glitches during transition which the external target may be sensitive to if it is not using an over-sampling scheme. To counter this, the SCL and SDA outputs from the internal state machine are flopped before they are emitted.

This adds a one cycle module clock delay to both signals. If the module clock is sufficiently faster than I2C line speeds (for example 24MHz), this is not an issue. However if the line speeds and the module clock speeds become very close (2x), the 1 cycle delay may have an impact, as the internal state machine may mistakenly think it has sampled an SDA that has not yet been updated.

Therefore, it is recommended that the internal module clock frequency is much higher than the line speeds. Another reason to have this higher internal clock frequency is that the timing parameters can be more accurately defined, which helps attain the desired I2C clock rate. Since there are currently also a few cycles discrepancy between the specified timings and the actual ones (as described in the [Programmer's Guide](https://github.com/lowRISC/opentitan/blob/master/hw/ip/i2c/doc/programmers_guide.md#timing-parameter-tuning-algorithm)), it is recommended that the internal module clock frequency is at least 24x higher than the I2C line speeds.