A2D converter note

- Resistors…

RS232 (Asynchronous Serial Protocol)

- Point-to-point wiring & protocol
  - Allows for bidirectional transmission (need two wires for this)
- No shared clock
  - Must have agreed on a transmission frequency ahead of time
  - Clocks are synchronized with a start bit
  - Transmission from A to B does not have to be synchronized with the transmission from B to A

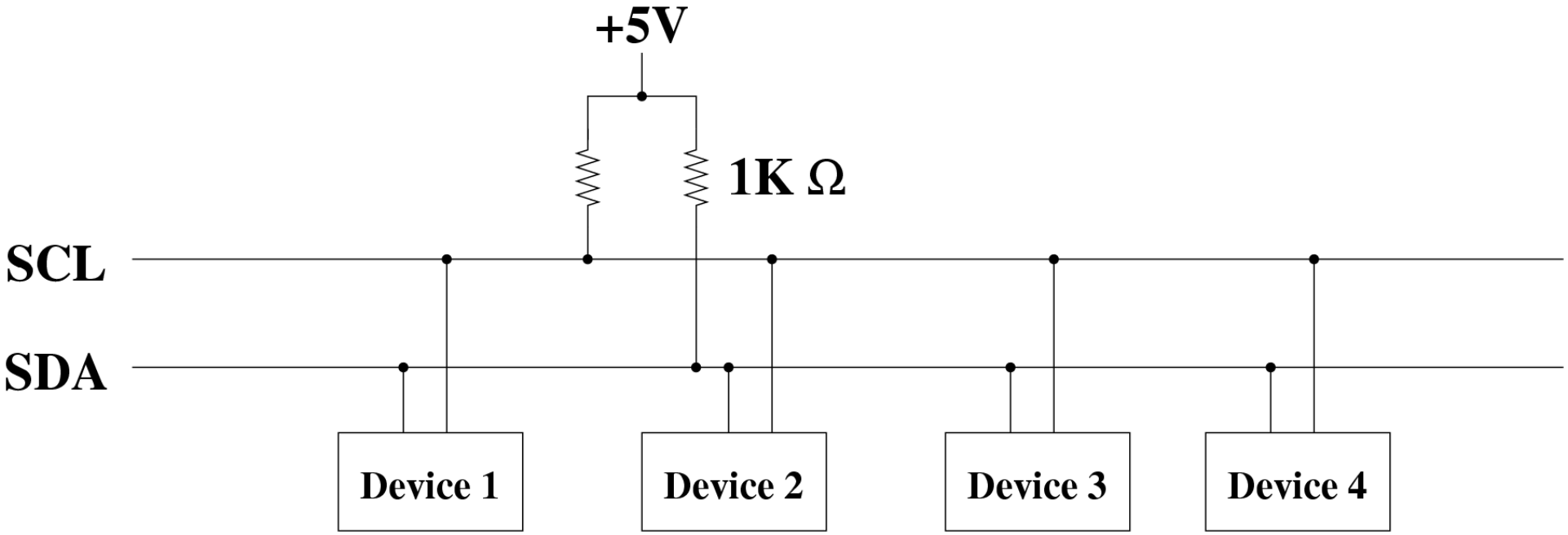
Synchronous Serial Protocols

- A clock signal is shared
  - No longer have to assume that two clocks stay in phase
  - Allows us to run the clock much faster than with asynchronous protocols
- Master/slave structure (for our purposes)
  - Master is responsible for producing the clock and any other control signals
  - Potential for many devices to share a common bus

Serial Communication with I^2C (Inter-Integrated Circuit)

- Potentially many masters
- Unidirectional communication: only one device may write to the bus at any one time
- Only two signal lines:
  - SDA: data transmission
  - SCL: clock
  - Some refer to this as a **Two Wire Interface**

I²C Device Interconnection



- Any device can be a master or a slave
- Devices drive lines low, but rely on the pull-up resistors to bring the lines high

I^2C Device Interconnection

Pull-up resistors

- Devices only drive the SCL/SDA lines low
- Otherwise, they allow the lines to float
  - So they are pulled up to high by the resistors
- This prevents a line from being driven high by one device and low by another device (which would create a short)

I^2C

What is missing?

I^2C

What is missing?

- We need some way to address the individual devices
- We need to deal with the bus arbitration problem
  - When two or more devices try to drive the bus at the same time…

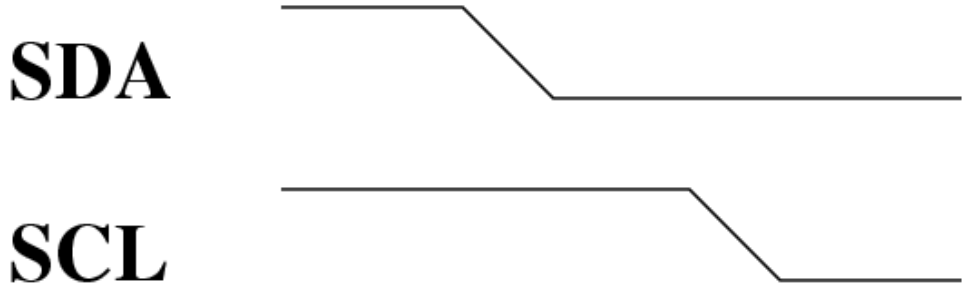
I^2C “Atoms”

3 conditions: start, data, and stop

- These atomic components make up larger packets

I^2C “Atoms”

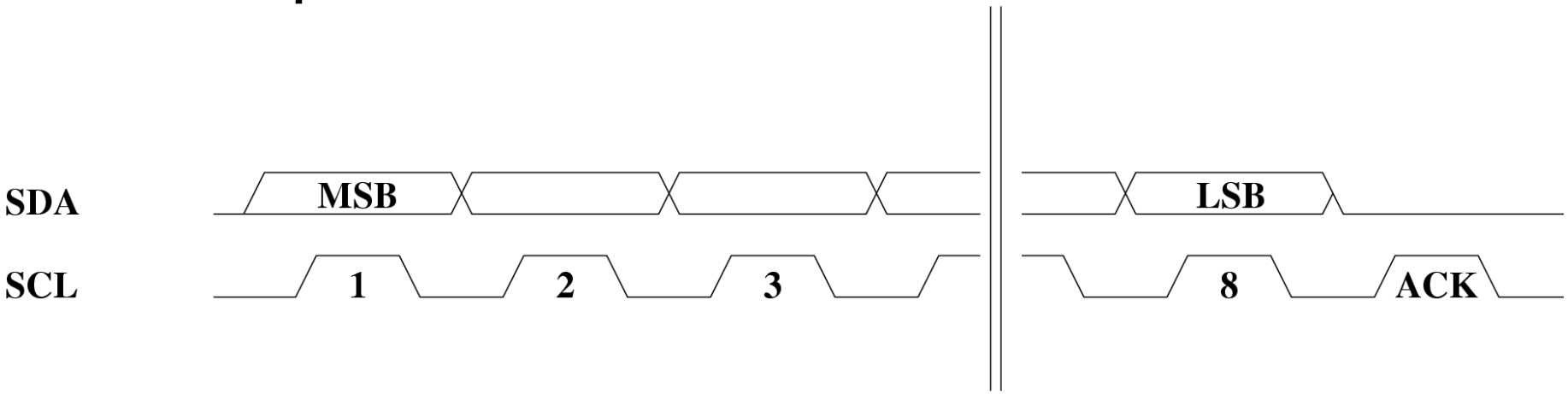
Start condition: SCL drop follows SDA drop



Once a master produces this sequence, it assumes that it “owns” the bus

I²C “Atoms”

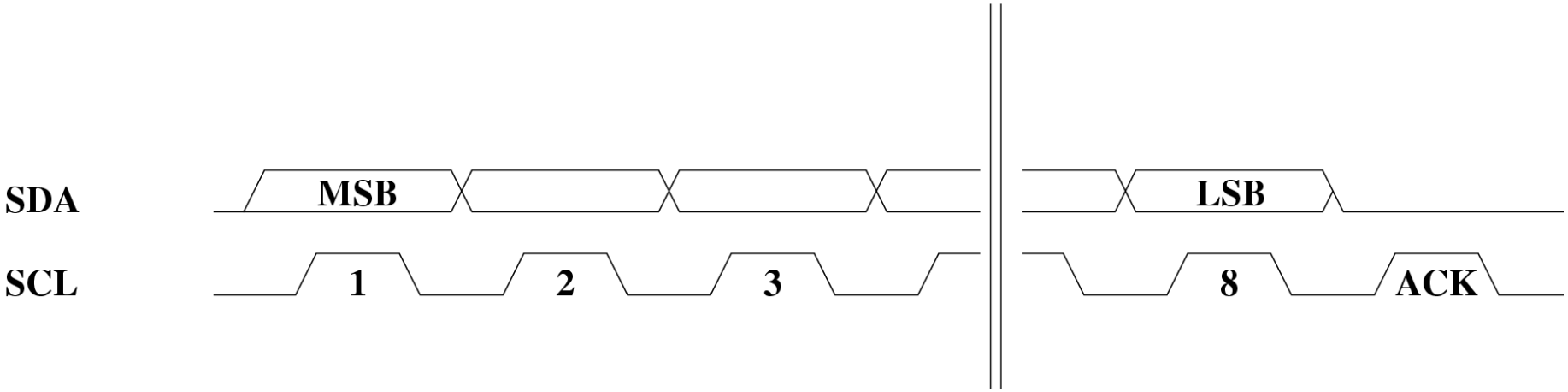
Data phase



- Data must be valid while the clock is high
- Bit is sampled on the rising edge of the clock

I²C “Atoms”

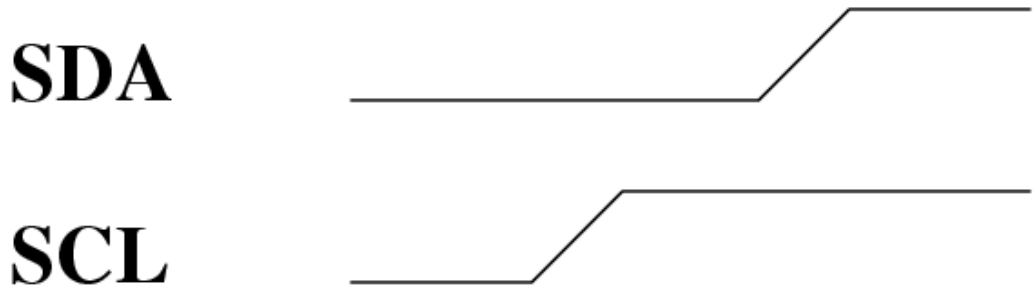
Data



- Master generates clock
- Sender (master or slave) generates data
- Receiver generates ACK bit (a low typically)

I²C “Atoms”

Stop Condition



- Order in which the pins return to high is important
- Master relinquishes control of the bus

Packets

Packets involve a multi-byte transfer between devices

- Start/stop
- Address
- Data

Addressing Devices

The first byte of any packet includes a device address.

- In the simple case, 7-bit addresses are used
  - These are the upper 7 bits of the byte
- The least significant bit indicates a read (1) or a write(0) to the addressed device
- The slave must ACK the address

Sending N Bytes from Master to Slave

How do we do it?

A Full Write Packet

The write packet includes the following:

- Start condition
- Address byte (with LSB=0)
- Data byte 1 (followed by an ACK=0 from the slave)
- Data byte 2 (followed by an ACK=0)

  :

- Data byte N (followed by an ACK=0)
- Stop condition

A Full Write Packet

- If at any point, the slave does not ACK a byte, this is considered an error and should be handled by the master

Slave Transmission to the Master

- Must be in response to a master request
- Master still generates the clock signal

Slave Transmission to the Master

If the slave is capable of multi-byte transfer, the master can control whether a next byte will be sent:

- If the master acknowledges the byte, then the device will send the next byte
- If the master leaves acknowledge bit high, then the device will not try to send another byte

Sending N Bytes from Slave to Master

What does the transaction look like?

A Full Read Packet

The read transaction includes the following:

- Start condition (driven by the master)
- Address byte (by master; with LSB=1)
- Data byte 1 (driven by the slave, but clock is driven by the master; ACK = 0 from the master)
- Data byte 2 (ACK = 0)

  :
- Data byte N (ACK = 1)
- Stop condition (by master)

Mixed Transactions

Some transactions will require both a write and a read

- For example: reading from an EEPROM device:
  - Master must send the command and addressing information
  - Then the master begins the read phase

Mixed Transactions

- In this case, the master will first generate a write packet and then a read packet
- But: there will not be an intermediate stop condition, instead a restart condition separates the two halves of the transaction
- This ensures that the master maintains control of the bus through the entire transaction

Mixed Transactions

What would this transaction look like?

Mixed Transactions

- Start condition
- Address byte (with LSB=0)
- Data byte 1 (followed by an ACK=0 from the slave)
- Data byte 2 (followed by an ACK=0)

  :
- Data byte N
- Restart condition
- Address byte (with LSB=1)
- Data byte 1 driven by slave (followed by an ACK=0 from the master)
- Data byte 2 (followed by an ACK=0)

  :
- Data byte M (followed by an ACK=1)
- Stop condition

Problem with Multiple Masters

- A master is not allowed to initiate a transaction if one is already in progress
  - We know this is the case if either SCK or SDL are low
- But – multiple masters may try to initiate at approximately the same time
  - How do we detect this?

Masters Colliding on the Bus

- When a master stops driving the line low, it can read the line state
- If the line does not return high, then we have a collision
- The master must abort its transaction and initiate at a later time
- The other master can continue its transaction without error

I^2C with 10-Bit Addresses

- More modern slave devices use 10-bit addressing
- This requires 2 bytes:
  - 11110XXD XXXXXXXX
    - Note that DEH is incorrect on this detail
  - X = address bit
  - D = direction (R/W)

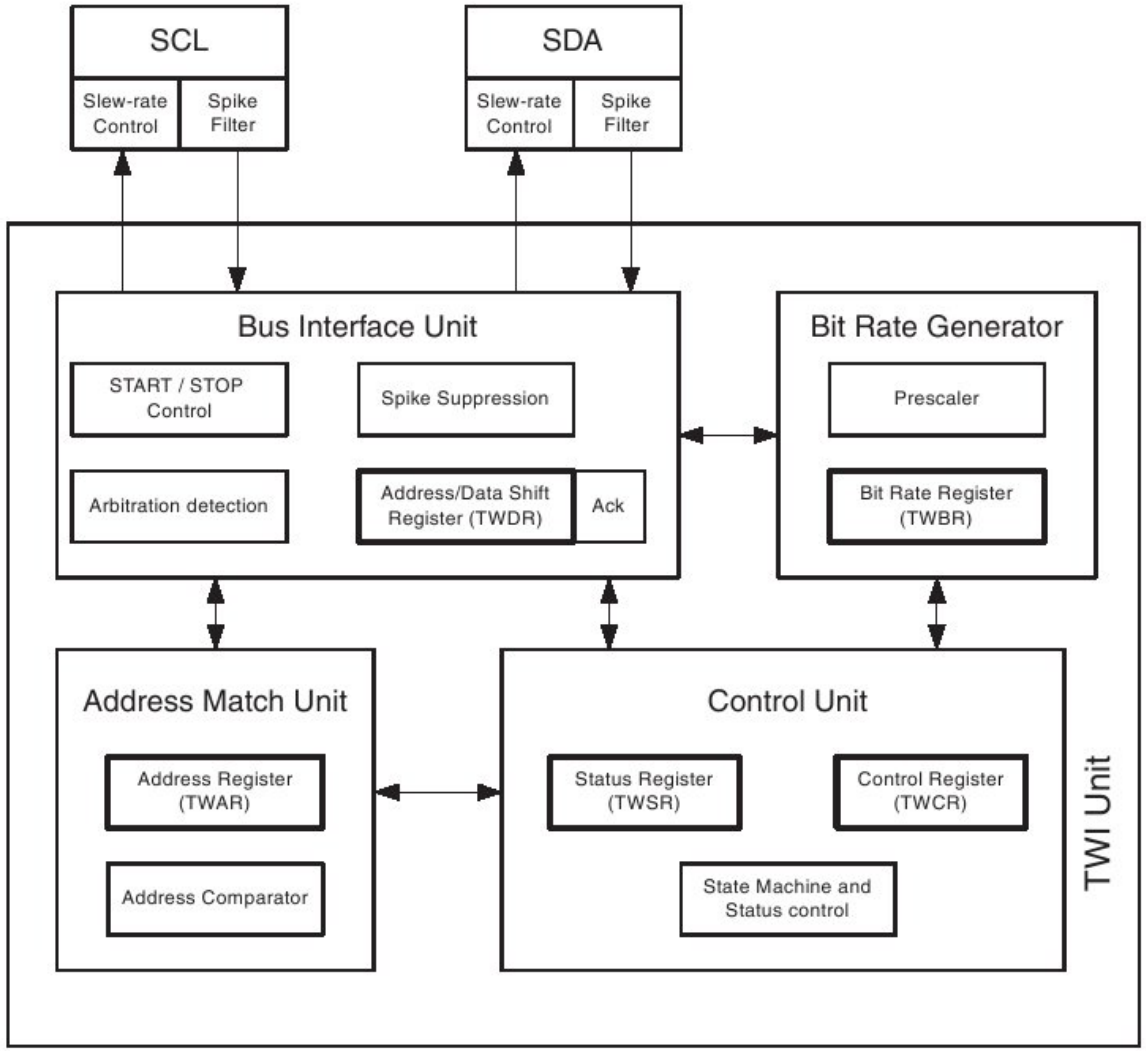
I^2C on the Mega8s

Support in hardware

- Software interacts at the level of start/stop conditions and single byte data/address transfers
- Can be configured as a master or a slave
- Slave: address is configurable
- Master: clock rate is configurable

- (details optional)

Mega8 “TWI” Hardware



Software Interaction with the TWI Hardware: Initiation

For each condition and data byte:

1. Configure the hardware by setting the appropriate register states
   - This includes committing to whether an ACK will be generated (by the receiver)
2. Enable the hardware for the transaction
   - Set the TWINT bit of TWCR

Software Interaction with the TWI Hardware: After the Transaction

1. TWINT bit of TWCR will go from low to high
   - Note that some bits of TWCR have multiple functions. So: read and write operations do not access the same registers

2. Can generate an interrupt on this event

OUlib Support

Provides (for the most part) a level of abstraction beyond the bit flipping

- Some functions will initiate a transaction automatically (this is documented in the code)

Protocol Example

Slave transmission of data to master:

- Byte 1: data byte 1
- Byte 2: data byte 2

  :

- Byte N: data byte N

Note: in this case, the master knows the number of bytes that will be coming from the slave

(most) Possible Status Codes for Master Receiver Case

Status codes in response to an interrupt (TWSR register)

From twi.h:

- TW_START: a start has been successfully executed on the bus.
- TW_REP_START: a restart (repeated start) has been executed
- TW_MR_ARB_LOST: this master has not been successful at claiming the bus
- TW_MR_SLA_ACK: a slave device has acknowledged being addressed
- TW_MR_SLA_NACK: a slave device has not acknowledged the address
- TW_MR_DATA_ACK: a slave device has transmitted data to the master and the master has acknowledged
- TW_MR_DATA_NACK: a slave device has transmitted data to the master and the master has not acknowledged
- TW_BUS_ERROR: an error has occurred

TWI Events: Master as Receiver

Table 67. Status codes for Master Receiver Mode

Status Code (TWSR) Prescaler Bits are 0	Status of the Two-wire Serial Bus and Two-wire Serial Interface Hardware	Application Software Response					Next Action Taken by TWI Hardware
		To/from TWDR	To TWCR				
			STA	STO	TWINT	TWEA	
0x08	A START condition has been transmitted	Load SLA+R	0	0	1	X	SLA+R will be transmitted ACK or NOT ACK will be received
0x10	A repeated START condition has been transmitted	Load SLA+R or	0	0	1	X	SLA+R will be transmitted ACK or NOT ACK will be received
		Load SLA+W	0	0	1	X	SLA+W will be transmitted Logic will switch to Master Transmitter mode
0x38	Arbitration lost in SLA+R or NOT ACK bit	No TWDR action or	0	0	1	X	Two-wire Serial Bus will be released and not addressed Slave mode will be entered
		No TWDR action	1	0	1	X	A START condition will be transmitted when the bus becomes free
0x40	SLA+R has been transmitted; ACK has been received	No TWDR action or	0	0	1	0	Data byte will be received and NOT ACK will be returned
		No TWDR action	0	0	1	1	Data byte will be received and ACK will be returned
0x48	SLA+R has been transmitted; NOT ACK has been received	No TWDR action or	1	0	1	X	Repeated START will be transmitted
		No TWDR action or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		No TWDR action	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x50	Data byte has been received; ACK has been returned	Read data byte or	0	0	1	0	Data byte will be received and NOT ACK will be returned
		Read data byte	0	0	1	1	Data byte will be received and ACK will be returned
0x58	Data byte has been received; NOT ACK has been returned	Read data byte or	1	0	1	X	Repeated START will be transmitted
		Read data byte or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		Read data byte	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset

Notes

- Macros for TWI events are defined by avr-LIBC
- The TWI events will only partially determine your FSM events
  - E.g., a byte counter
- Similar event lists for other modes:
  - Master as transmitter
  - Slave as transmitter
  - Slave as receiver

Master Receiver

What is the FSM?

- States?
- Events?
- Actions?

Master Receiver

- States: tell us where we are in the protocol
- Events:
  - TWI status
  - other variables (we will need a counter)
- Actions:
  - I^2C atomic operations (start, stop, data, restart)
  - Manipulate internal variables

Master Receiver

What does the FSM look like?

(most) Possible Status Codes for Master Transmitter Case

Status codes in response to an interrupt (TWSR register)

From twi.h:

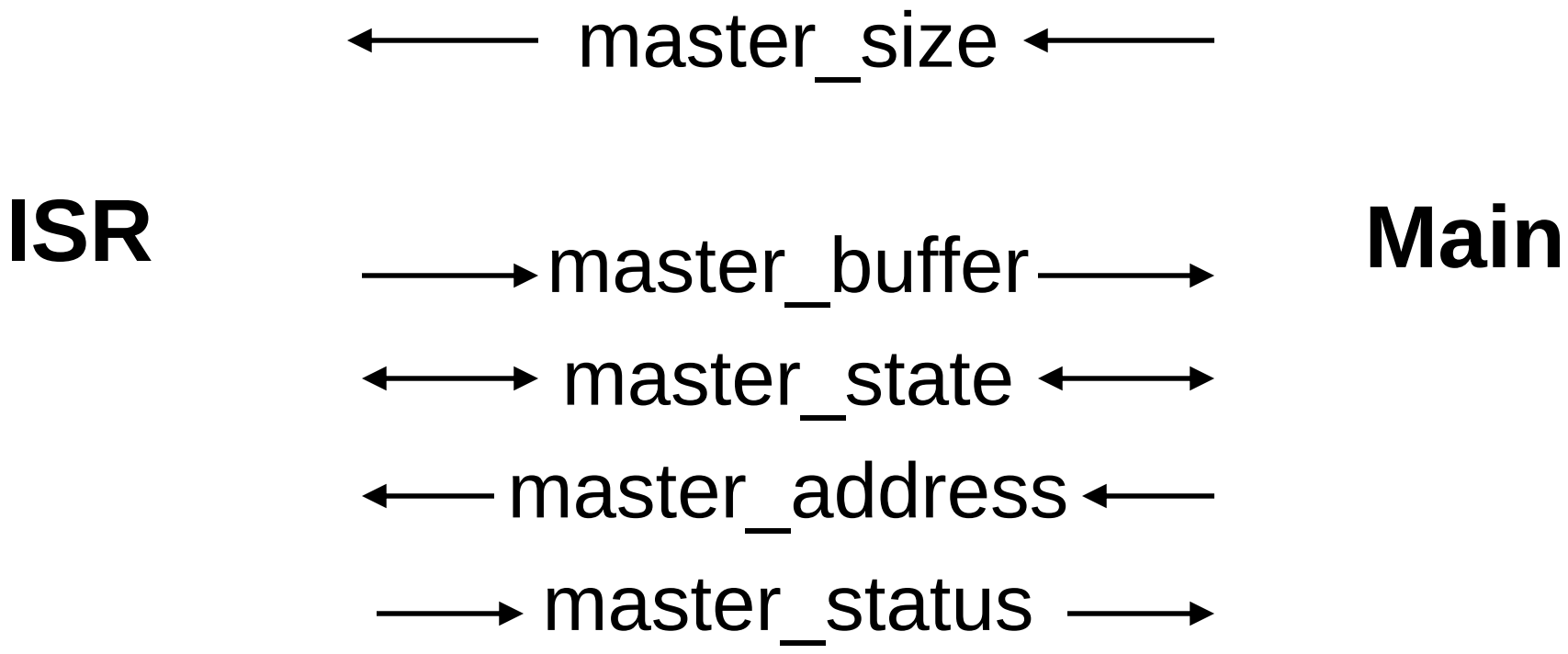
- TW_START
- TW_REP_START
- TW_MR_ARB_LOST
- TW_MR_SLA_ACK: a slave has acknowledged being addressed
- TW_MR_SLA_NACK: no slave has acknowledged
- TW_MR_DATA_ACK: data transmitted to slave has been acknowledged
- TW_MR_DATA_NACK : data sent, but not acknowledged
- TW_BUS_ERROR: an error has occurred

TWI Events: Master as Transmitter

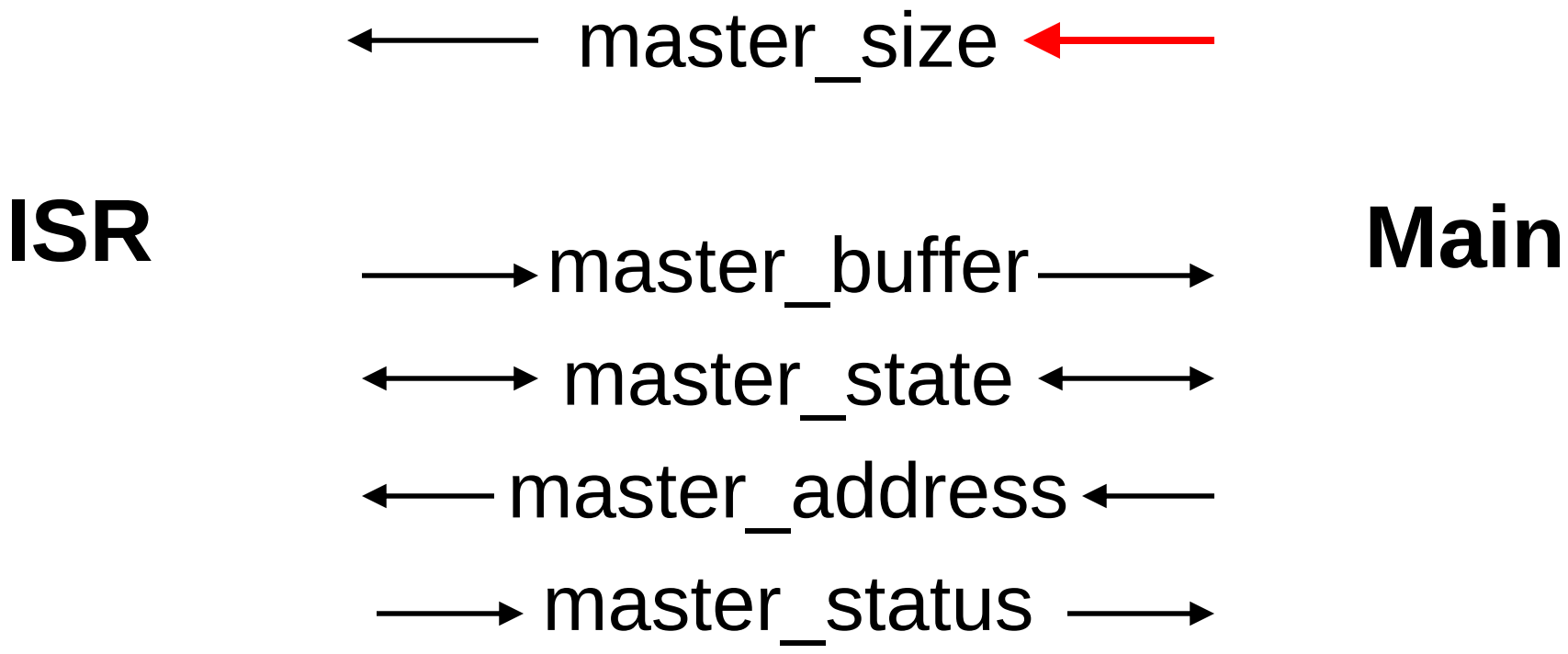
Table 66. Status codes for Master Transmitter Mode

Status Code (TWSR) Prescaler Bits are 0	Status of the Two-wire Serial Bus and Two-wire Serial Interface Hardware	Application Software Response					Next Action Taken by TWI Hardware
		To/from TWDR	To TWCR				
			STA	STO	TWINT	TWEA	
0x08	A START condition has been transmitted	Load SLA+W	0	0	1	X	SLA+W will be transmitted; ACK or NOT ACK will be received
0x10	A repeated START condition has been transmitted	Load SLA+W or	0	0	1	X	SLA+W will be transmitted; ACK or NOT ACK will be received
		Load SLA+R	0	0	1	X	SLA+R will be transmitted; Logic will switch to Master Receiver mode
0x18	SLA+W has been transmitted; ACK has been received	Load data byte or	0	0	1	X	Data byte will be transmitted and ACK or NOT ACK will be received
		No TWDR action or	1	0	1	X	Repeated START will be transmitted
		No TWDR action or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		No TWDR action	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x20	SLA+W has been transmitted; NOT ACK has been received	Load data byte or	0	0	1	X	Data byte will be transmitted and ACK or NOT ACK will be received
		No TWDR action or	1	0	1	X	Repeated START will be transmitted
		No TWDR action or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		No TWDR action	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x28	Data byte has been transmitted; ACK has been received	Load data byte or	0	0	1	X	Data byte will be transmitted and ACK or NOT ACK will be received
		No TWDR action or	1	0	1	X	Repeated START will be transmitted
		No TWDR action or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		No TWDR action	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x30	Data byte has been transmitted; NOT ACK has been received	Load data byte or	0	0	1	X	Data byte will be transmitted and ACK or NOT ACK will be received
		No TWDR action or	1	0	1	X	Repeated START will be transmitted
		No TWDR action or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		No TWDR action	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x38	Arbitration lost in SLA+W or data bytes	No TWDR action or	0	0	1	X	Two-wire Serial Bus will be released and not addressed Slave mode entered
		No TWDR action	1	0	1	X	A START condition will be transmitted when the bus becomes free

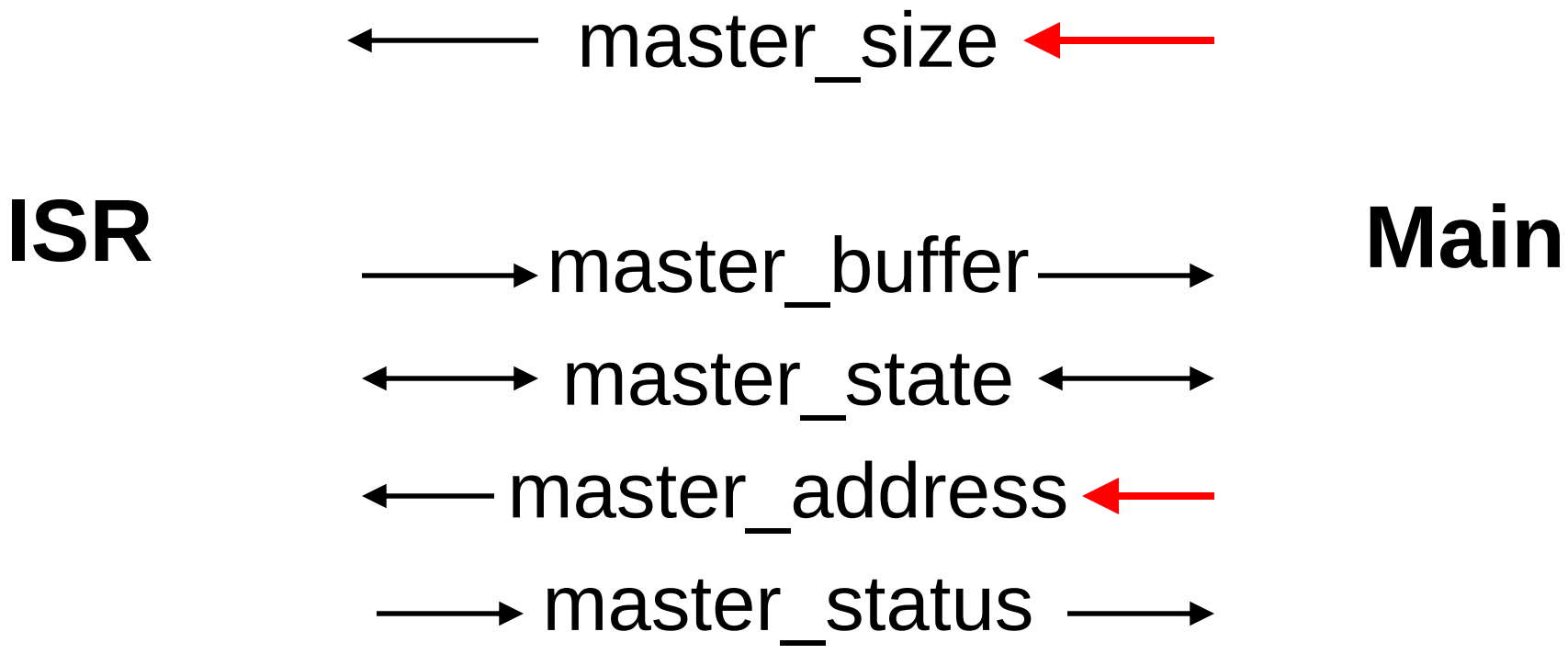
ISR/Main Program Communication



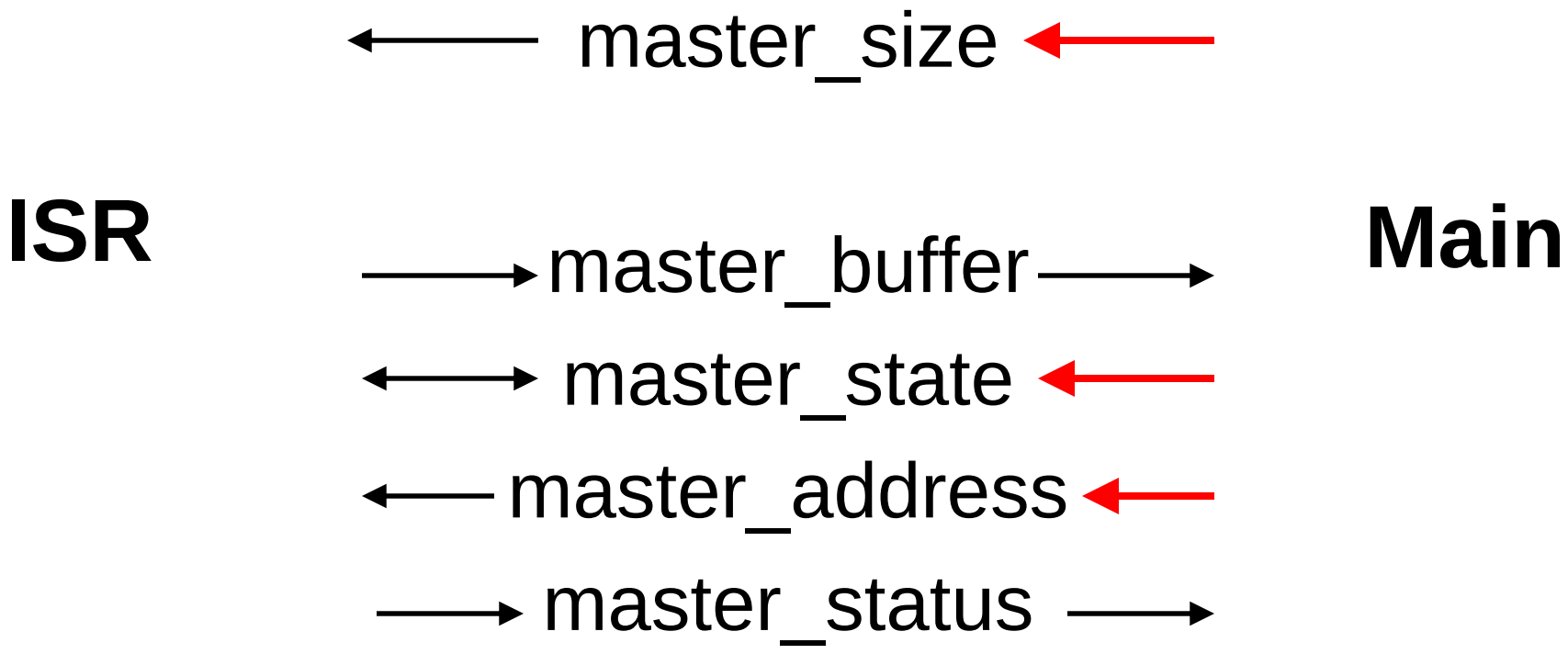
ISR/Main Program Communication



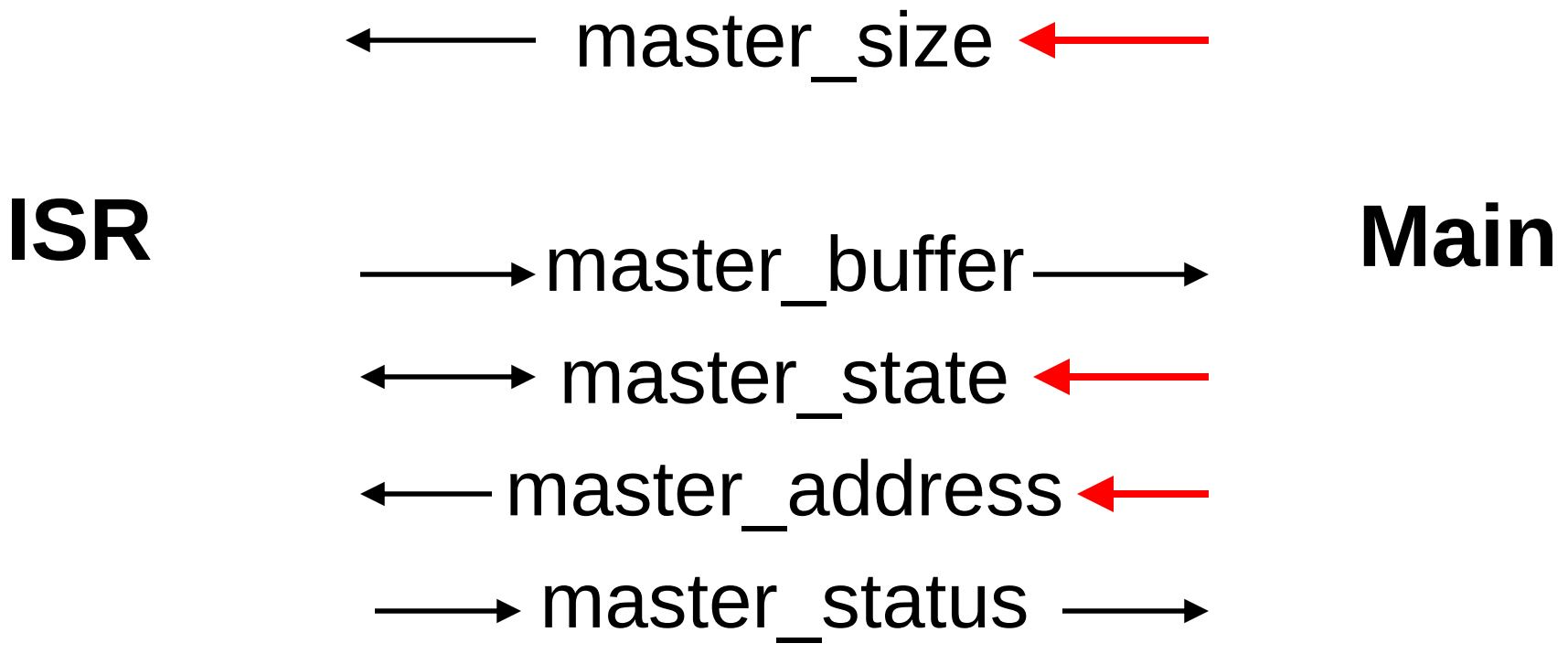
ISR/Main Program Communication



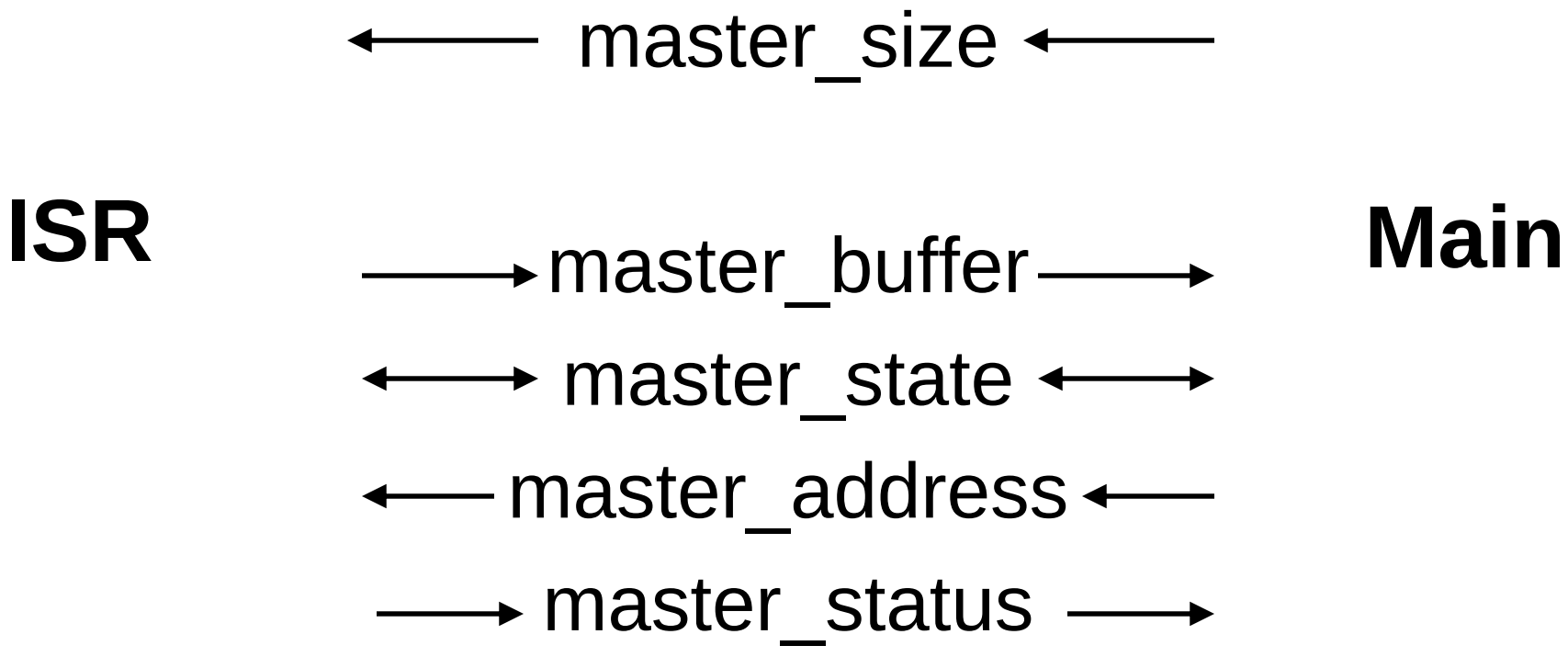
ISR/Main Program Communication



ISR/Main Program Communication

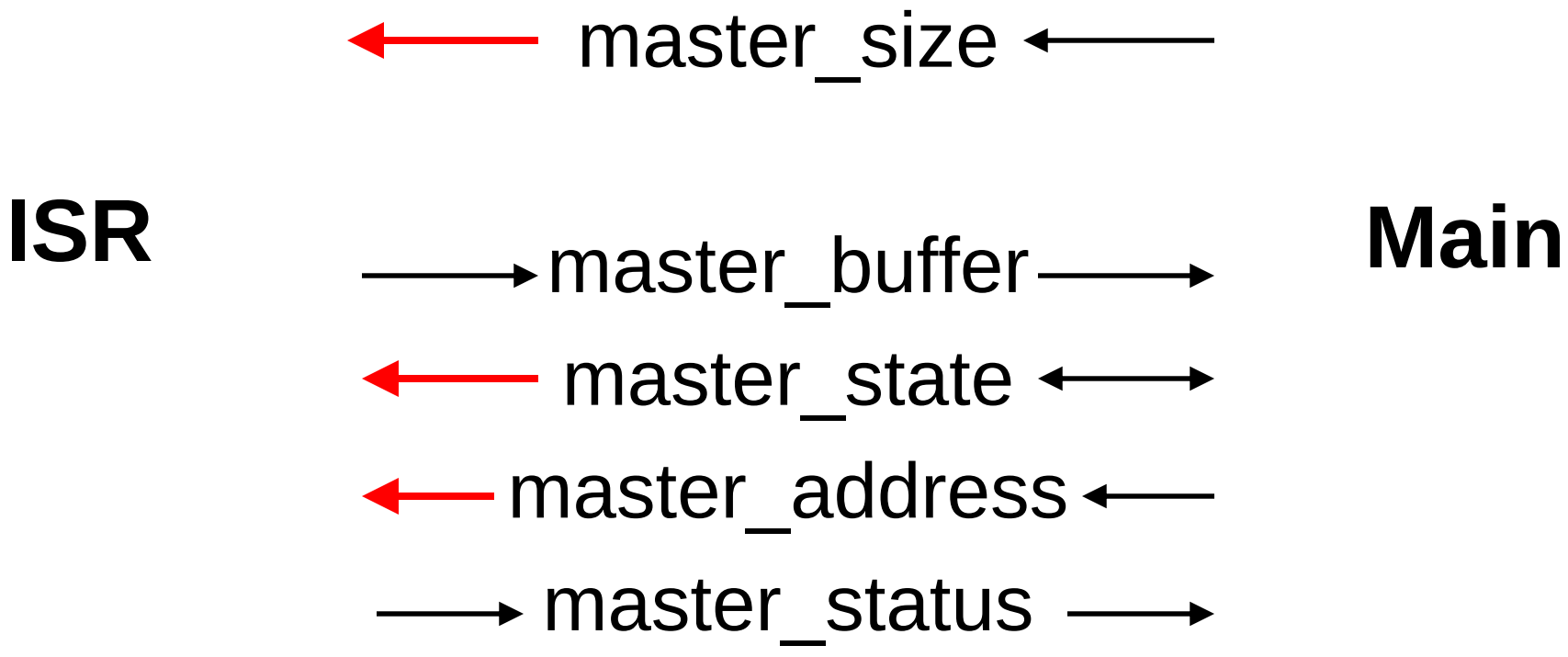


ISR/Main Program Communication



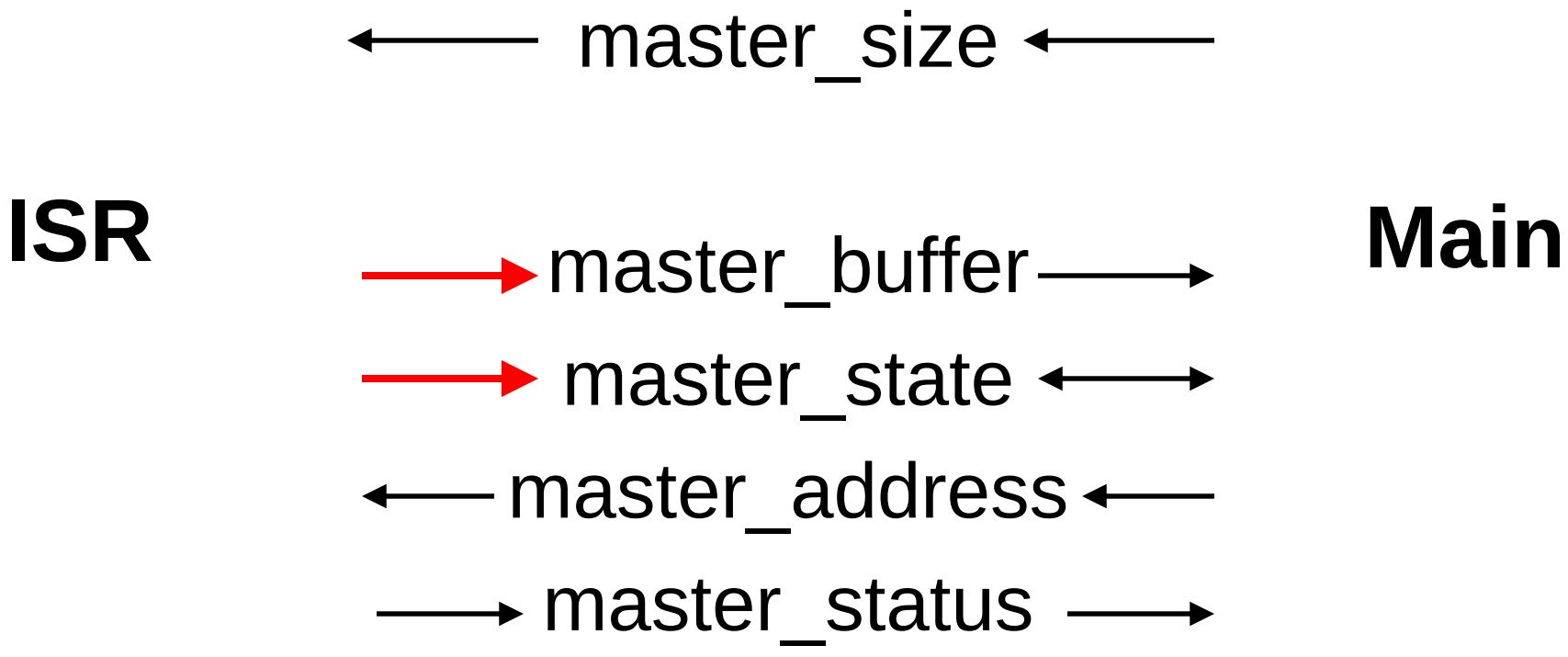
START Received: proceed with remaining transaction

ISR/Main Program Communication



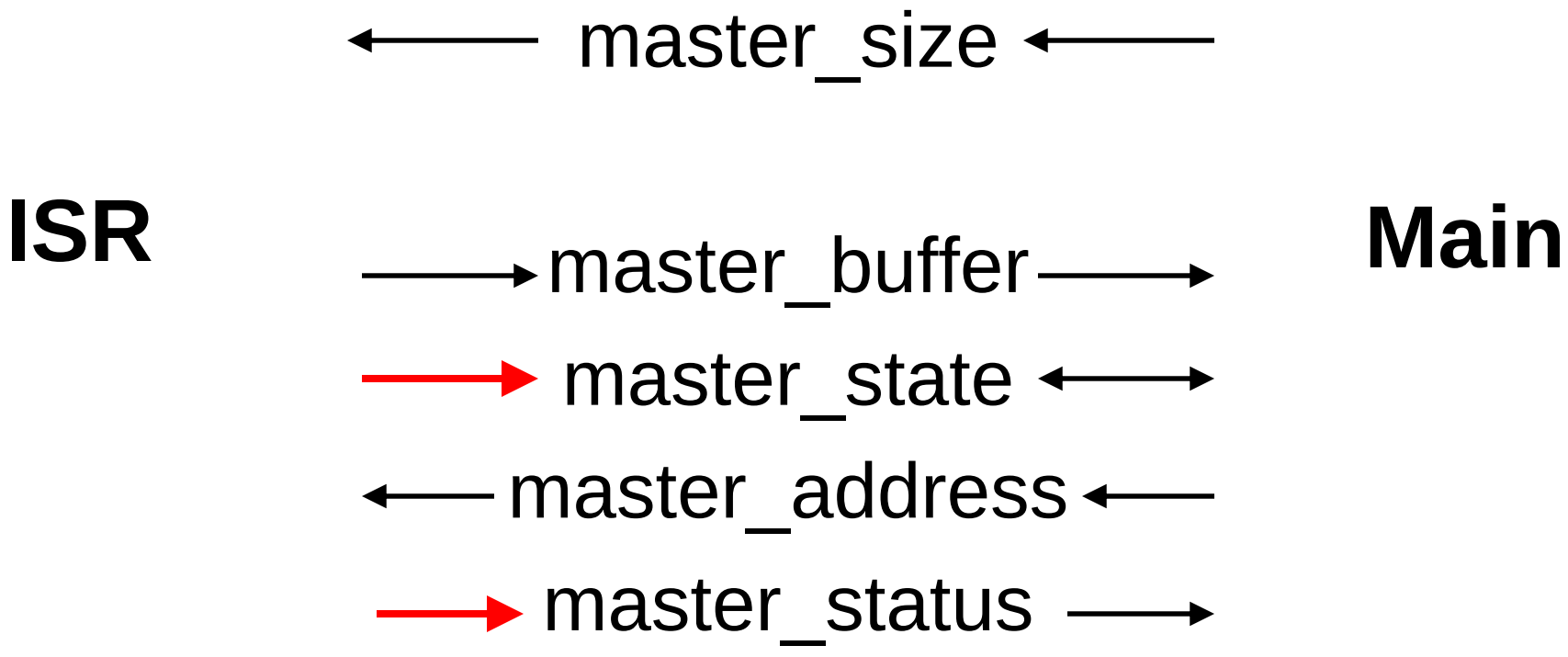
START Received: proceed with remaining transaction

ISR/Main Program Communication



During transaction

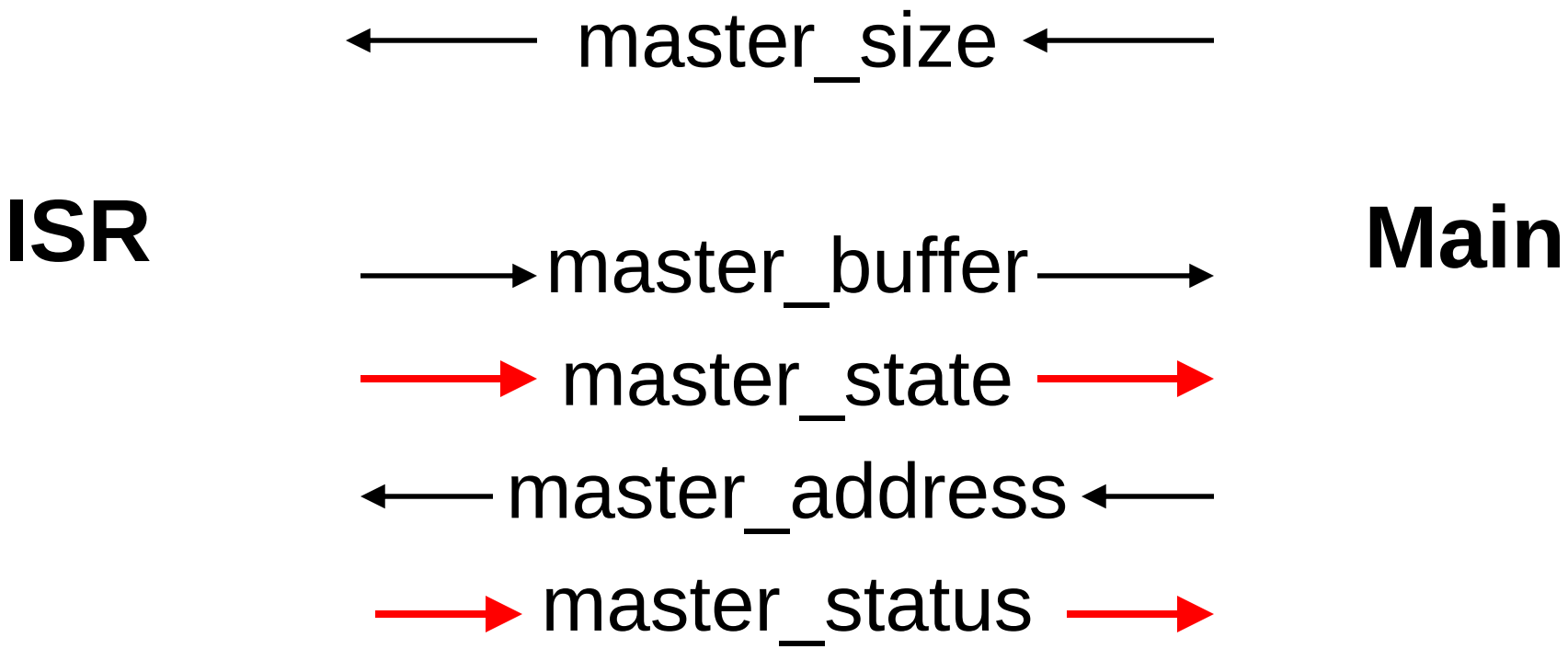
ISR/Main Program Communication



End Transaction

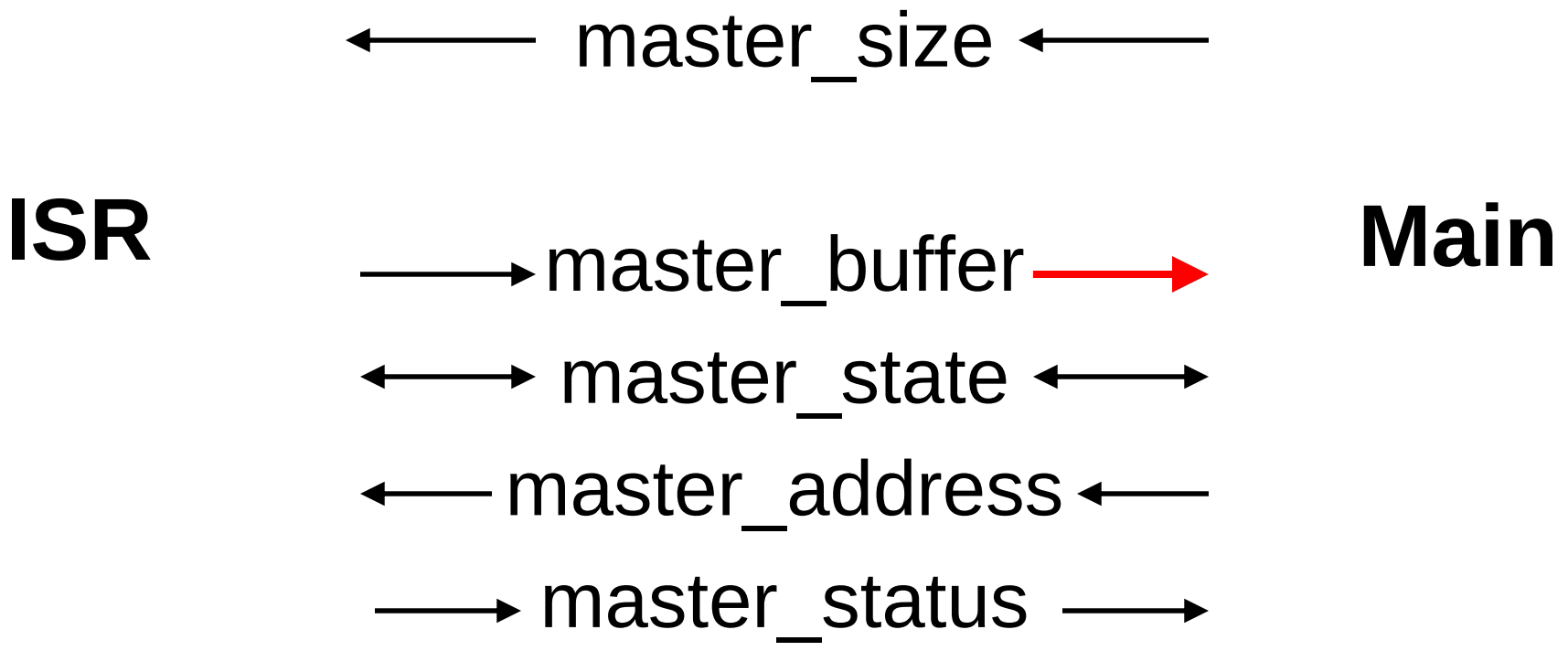
State is DONE or ERROR

ISR/Main Program Communication



Main "sees" state transition To DONE or ERROR

ISR/Main Program Communication



If DONE, extract buffer contents

An ISR Implementation

```
// FSM state
volatile int8_t master_state;

// Number of bytes that the master will receive
volatile uint8_t master_size;

// Buffer that the data is being dropped into
volatile uint8_t master_buffer[BUFFER_SIZE];

// Address of the slave
volatile uint8_t master_address;

// Current byte count
uint8_t master_counter;

// Copy of the status register (useful for error
   processing)
volatile uint8_t master_status;
```

An ISR Implementation

```
ISR(TWI_vect) {
  uint8_t status;
    switch(master_state)
      {
      case I2C_MASTER_STATE_WSTART:
         // We were waiting for the start to be asserted
         status = twi_get_status();
         if(status == TW_START) {
                  // Send slave address; move on to the next state
                  master_state = I2C_MASTER_STATE_WADDRESS;
                  twi_send_byte((master_address & 0xfe) | 0x1);
         }else{
                  // An error has occurred
                  master_state = I2C_MASTER_STATE_ERROR;
                  master_status = status;
                  // We assume that we lost arbitration, so do nothing to
                  //   the hardware
         };
         break;
```

Waiting for Address

```
case I2C_MASTER_STATE_WADDRESS:
   status = twi_get_status();
   if(status == TW_MR_SLA_ACK) {
     // The slave device acknowledged
     // We will positively acknowledge the next byte
     twi_set_ack(TWI_ACKNOWLEDGE_ENABLE);
     master_state = I2C_MASTER_STATE_WDATA;
     master_counter = 0;
     twi_set_twint();   // Force twint flag to be cleared:
                        // initiate read transaction

   }else{
     // An error has occurred
     master_state = I2C_MASTER_STATE_ERROR;
     twi_send_stop();
     master_status = status;
   };
   break;
```

Waiting for Data

```
case I2C_MASTER_STATE_WDATA:
  // We are waiting for <master_size> bytes of data
  status = twi_get_status();
  if(status == TW_MR_DATA_NACK) {
    // We did not ACK the last byte
     //   (so it was the last one)
    master_buffer[master_counter] = twi_get_byte();

    // We have received all of the bytes
    master_state = I2C_MASTER_STATE_DONE;
    twi_send_stop();
    master_status = status;
  }else if(status == TW_MR_DATA_ACK) {
```

Waiting for Data

```
case I2C_MASTER_STATE_WDATA:
  // We are waiting for <master_size> bytes of data
  status = twi_get_status();
  if(status == TW_MR_DATA_NACK) {
    :
  }else if(status == TW_MR_DATA_ACK) {
    // This is not the last byte
    master_buffer[master_counter] = twi_get_byte();
    // Increment byte count
    ++master_counter;

    if(master_counter == master_size-1) {
      // This is going to be the last byte to be received
      twi_set_ack(TWI_ACKNOWLEDGE_DISABLE);
    }else{
      // This is not the last byte
      twi_set_ack(TWI_ACKNOWLEDGE_ENABLE);
    };
    twi_set_twint();
  }else{
    // An error has occurred
```

Subtleties

- We transition to the DONE state as we are issuing the stop condition
- But: we don’t wait for the stop condition to be completed
- There is a danger that we will initiate the next transaction before the previous one is really complete

Subtleties

For some reason, there does not seem to be an interrupt generated for master receive mode when the stop condition is completed …

Solution:

- Before the main program is allowed to initiate the next transmission, it must check that the stop has occurred:

  twi_get_stop() will be true as long as we are still waiting for the stop to complete (so busy wait on this condition)

Subtleties

A common problem: the interrupt routine behaves as if it is never being called

- Check 1: is the interrupt being enabled?
- Check 2: do you have pull-up resistors on the SDA and SCL lines?

I^2C Protocol for the Major Component

- Need to handle both transmission to and reception from the slave
- We will use the mixed protocol as already described in the above example

- You may assume that the master knows *a priori* how many bytes will be transferred (N) and how many will be received (M)
- But – the master should properly handle the case where N= 0 or M=0

I^2C Compass

- Compass exposes a set of registers that can be read from and written to
- Writing to the compass: standard write I^2C transaction
- Reading from the compass: mixed I^2C transaction:
  - Send a register address
  - Start reading bytes

Debugging

- Use LEDs to indicate the FSM state
  - Help you to determine where your FSM gets hung
- Use Oscilloscopes
  - 2 channels possible: use for SDA and SCL
  - Learn how to capture and store traces
- Incremental implementation and debugging

On to SPI …