Serial Communications

Overview

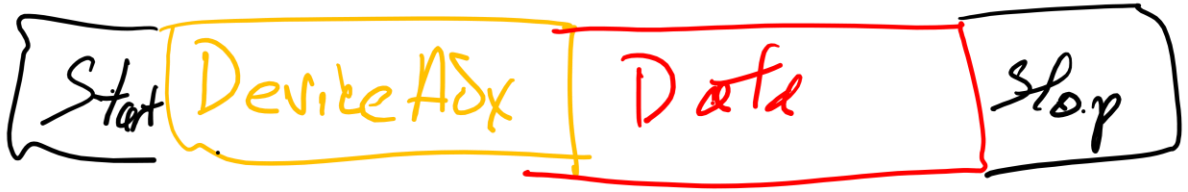
- Serial communications
  - Concepts
  - Tools
  - Software: polling, interrupts and buffering
- UART communications
  - Concepts
  - KL25 I2C peripheral
- SPI communications
  - Concepts
  - KL25 SPI peripheral
- I^2C communications
  - Concepts
  - KL25 I2C peripheral

Evolution of Communications for ES

- Go serial instead of parallel
  - Why? Fewer signals -> smaller PCBs, ICs, connectors… (but lower throughput)
  - How?
    - Timing reference: clock vs. no clock
    - Message framing: start and stop
    - Error control: detection, correction, retry
    - Flow control
- Go half-duplex
  - Why? Fewer signals -> smaller (but lower throughput)

Evolution of Communications for ES

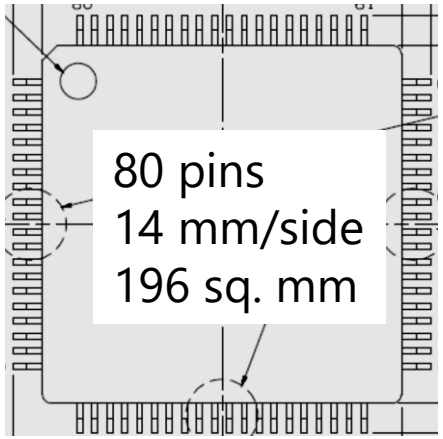
- Share bus ("medium") instead of using dedicated links
- Why? Smaller, share data more easily (but lower throughput)
- How?
  - Use access control to arbitrate access (MAC)
    - Collisions: Detection, prevention, avoidance, arbitration
  - Addressing to support multiple devices
    - In-message addressing vs. chip select lines
    - Addressing methods: per device, message type

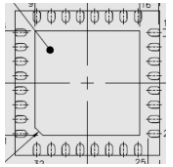


Why Communicate Serially?

- Although native word size for CPU is 32 bits, sending all of a word's bits simultaneously has disadvantages:
  - **Cost and weight:** larger IC package, more wires, larger connectors
  - **Mechanical reliability:** more wires => more connector contacts to fail
  - **Timing complexity:** some bits may arrive later than others due to variations in capacitance and resistance across conductors
  - **Circuit complexity and power:** may not want to have 16 different transmitters + receivers in the system
- Communicating serially reduces number of signals needed

Shrinking Packages for NXP MCUs

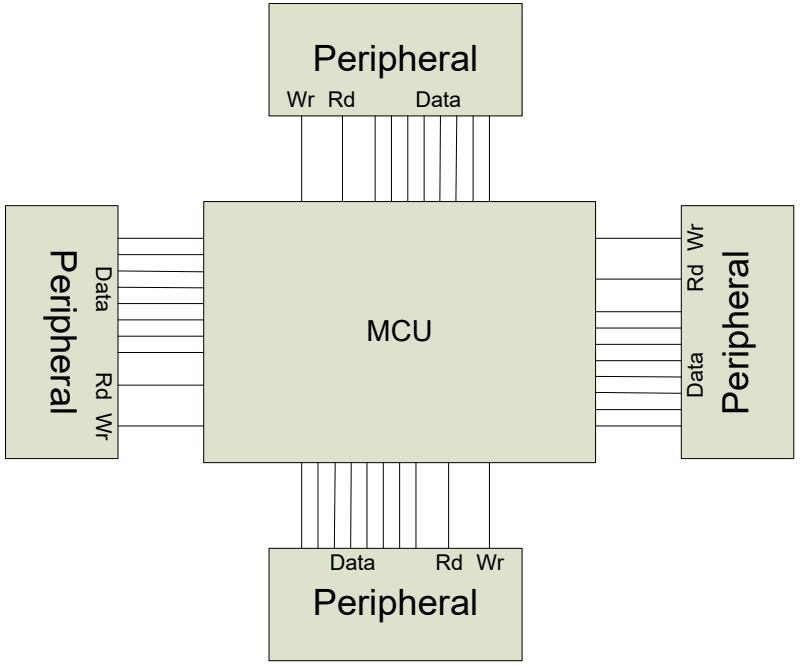




32 pins
5 mm/side
25 sq. mm

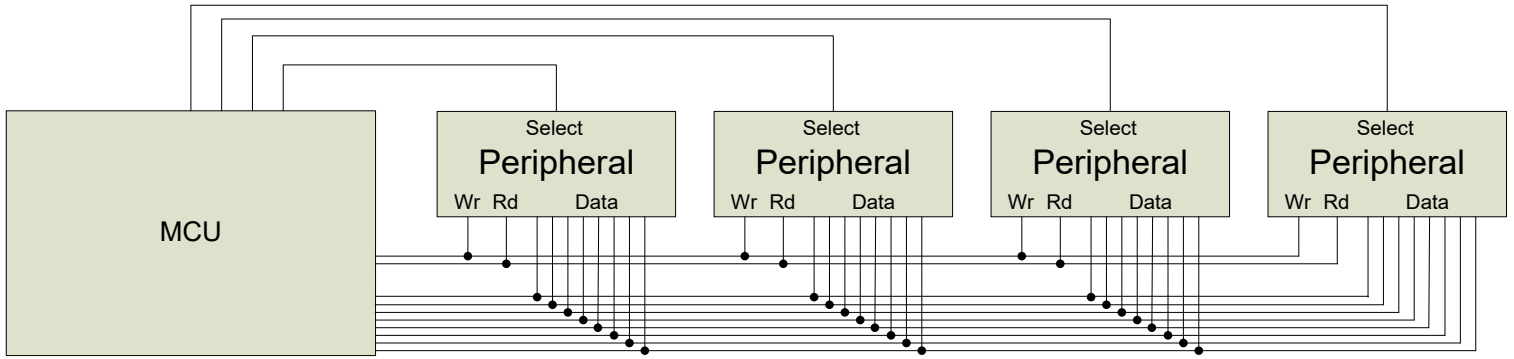
20 pins
1.94 mm/side
3.76 sq. mm

Example System



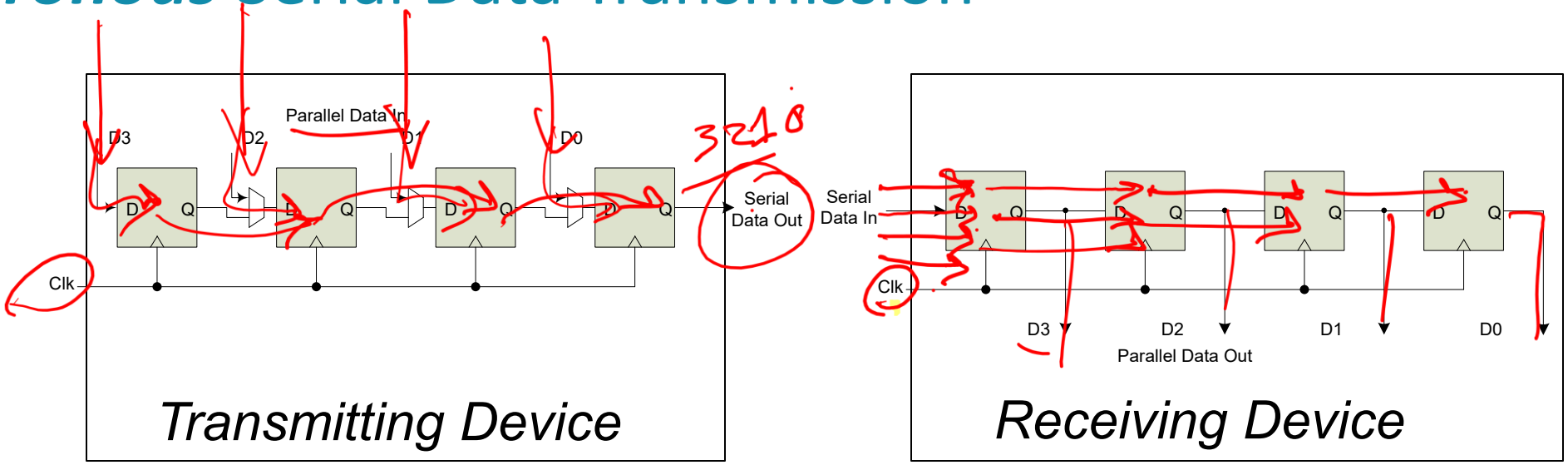
- Dedicated point-to-point connections
  - Parallel data lines, read and write lines between MCU and each peripheral
- Fast, allows simultaneous transfers
- Requires many connections, PCB area, scales badly
  - Need 4*(8+2) = 40 pins on MCU to communicate!

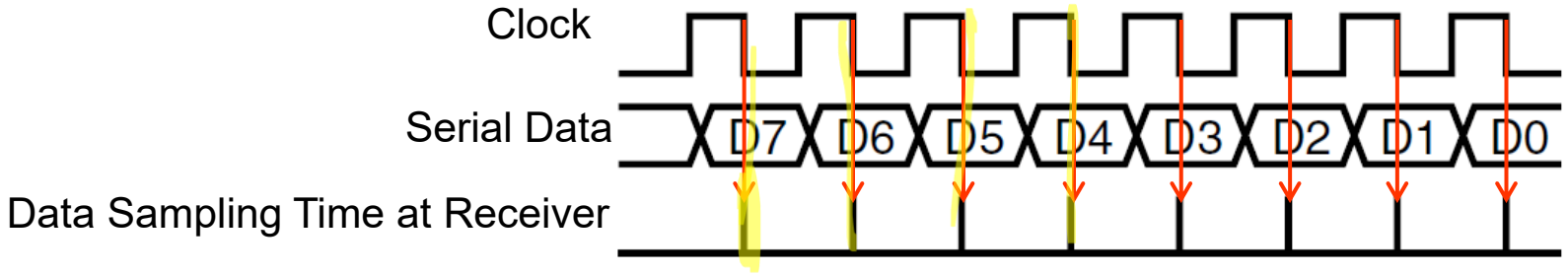
Parallel Buses



- All devices use buses to share data, read and write signals
- MCU uses individual select lines to address each peripheral
- MCU requires fewer pins for data, but still one per data bit
  - Need 4 + (8+2) = 14 pins on MCU to communicate
- MCU can communicate with only one peripheral at a time

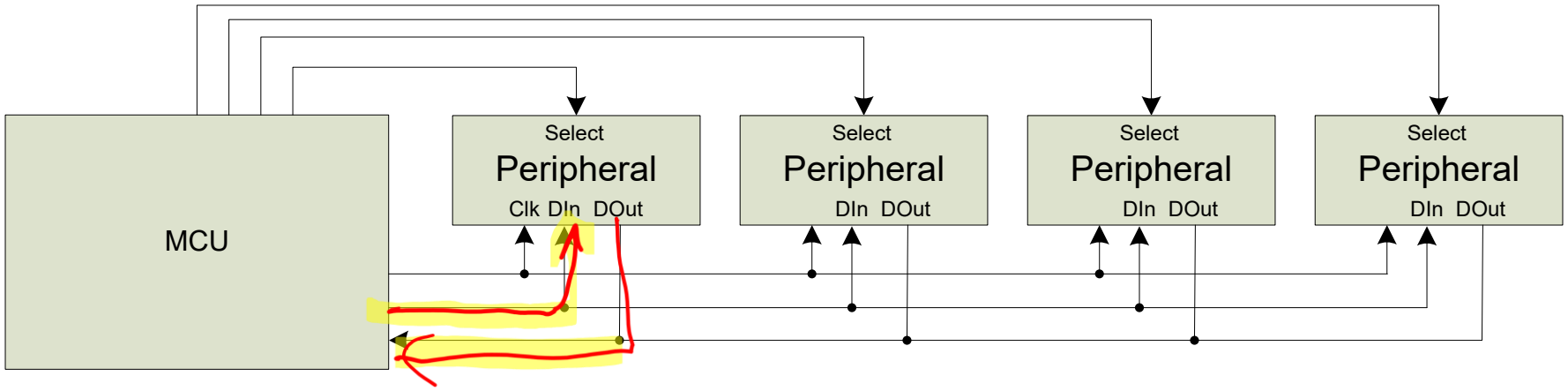
Synchronous Serial Data Transmission





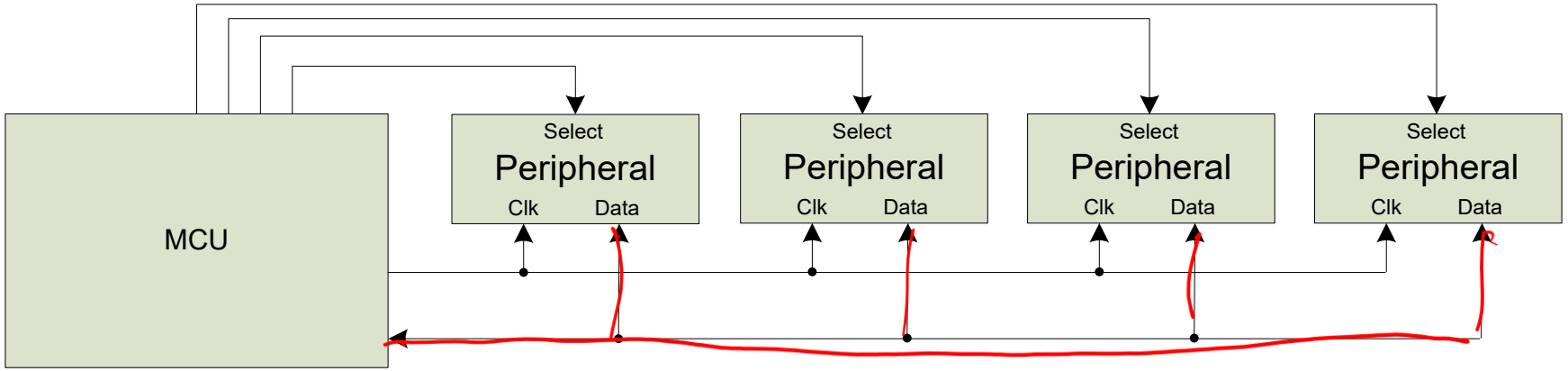
- Use shift registers and a clock signal to convert between serial and parallel formats
- Synchronous: an explicit clock signal is sent along with the data signal

Synchronous Full-Duplex Serial Data Bus



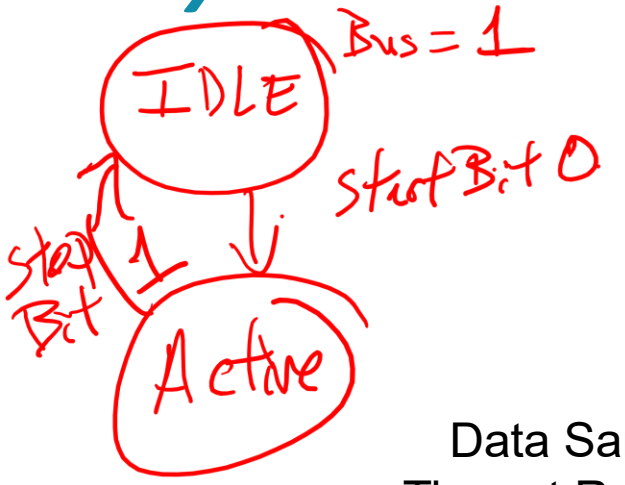
- Now can use two serial data lines - one for reading, one for writing.
  - Allows simultaneous send and receive *full-duplex communication*
  - Need 4 + 3 = 7 pins on MCU to communicate

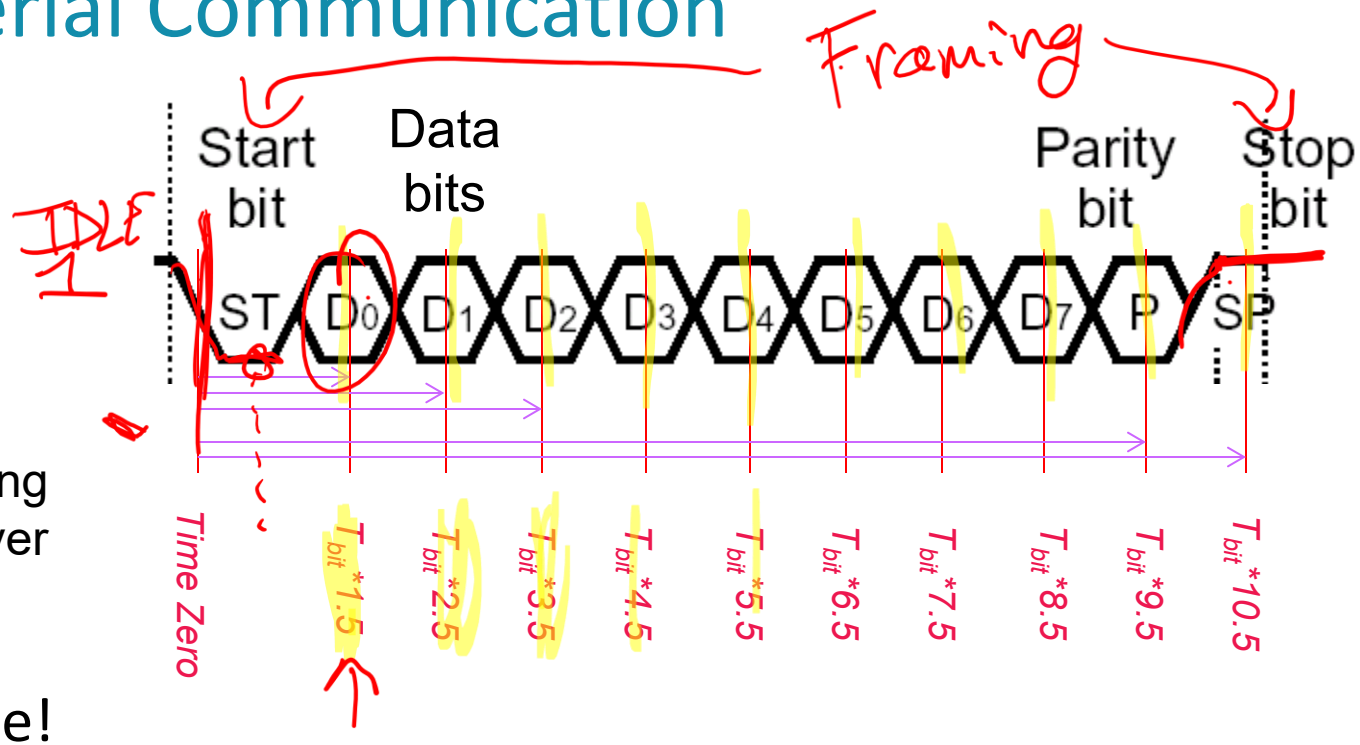
Synchronous Half-Duplex Serial Data Bus



- **Share the serial data line**
  - Need 4 + 2 = 6 pins on MCU to communicate
- **Doesn’t allow simultaneous send and receive - is *half-duplex communication***

Asynchronous Serial Communication



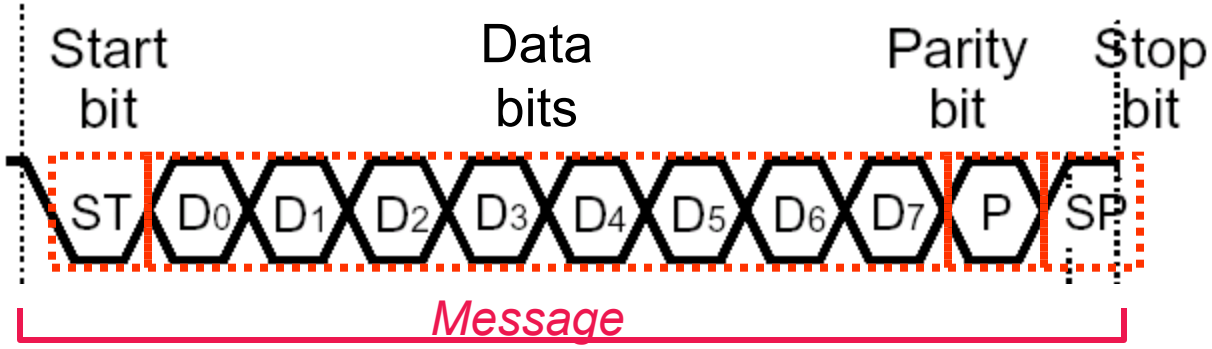


Data Sampling Time at Receiver

- Eliminate the clock line!
- Transmitter and receiver must generate clock *locally*
- States: Idle, Message
- Transmitter must add start bit (always same value) to indicate transition from Idle to message
- Receiver detects leading edge of start bit, then uses it as a timing reference for sampling data line to extract each data bit N at time $T_{bit}*(N+1.5)$
- Stop bit is also used to detect some timing errors

Serial Communication Specifics

- Data frame fields
  - Start bit (one bit)
  - Data (LSB or MSB first, and size – 7, 8, 9 bits)
  - Optional parity bit is used to make total number of ones in data even or odd
  - Stop bit (one or two bits)



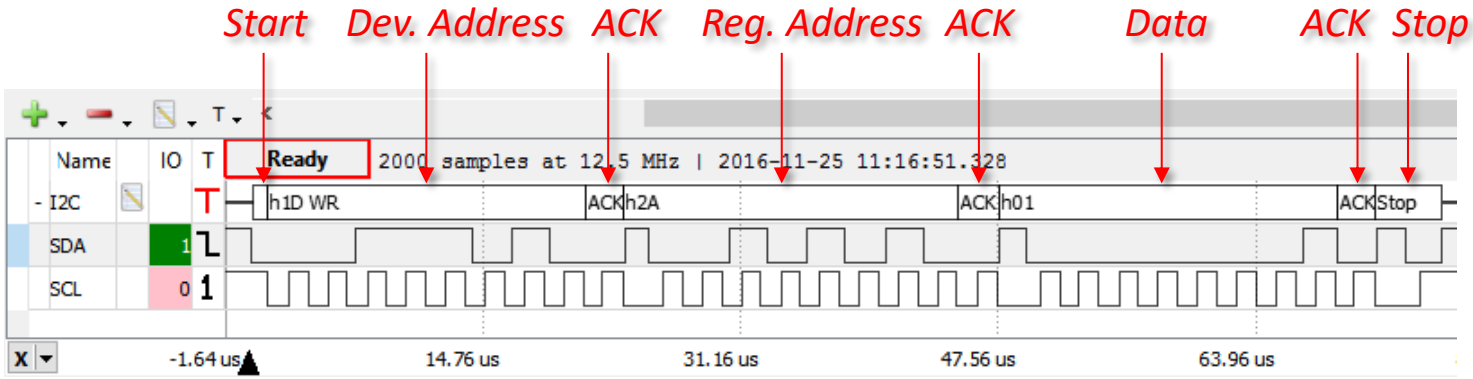
- All devices must use the same communications parameters
  - E.g. communication speed (300 baud, 600, 1200, 2400, 9600, 14400, 19200, etc.)
- Sophisticated network protocols have more information in each data frame
  - Medium access control – when multiple nodes are on bus, they must arbitrate for permission to transmit
  - Addressing information – for which node is this message intended?
  - Larger data payload
  - Stronger error detection or error correction information
  - Request for immediate response (“in-frame”)

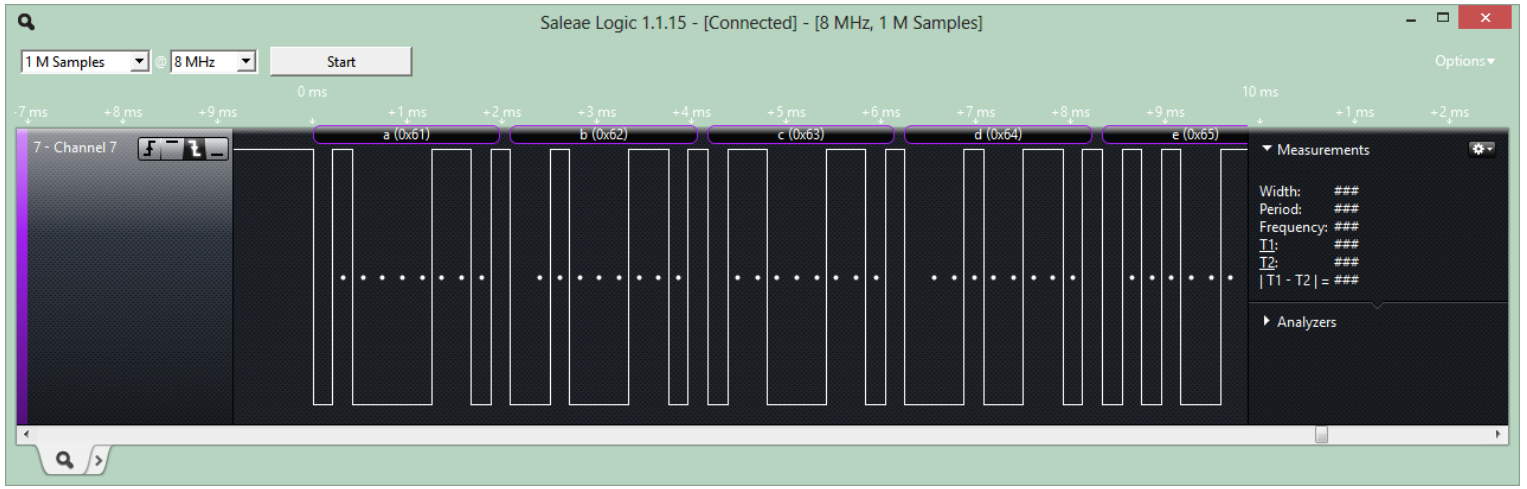
Error Detection

- Can send additional information to verify data was received correctly
- Need to specify which parity to expect: even, odd or none.
- Parity bit is set so that total number of “1” bits in data and parity is even (for even parity) or odd (for odd parity)
  - 01110111 has 6 “1” bits, so parity bit will be 1 for odd parity, 0 for even parity
  - 01100111 has 5 “1” bits, so parity bit will be 0 for odd parity, 1 for even parity
- Single parity bit detects if 1, 3, 5, 7 or 9 bits are corrupted, but doesn’t detect an even number of corrupted bits
- Stronger error detection codes (e.g. Cyclic Redundancy Check) exist and use multiple bits (e.g. 8, 16), and can detect many more corruptions.
  - Used for CAN, USB, Ethernet, Bluetooth, etc.

Tools for Serial Communications Development

- Tedious and slow to debug serial protocols with just an oscilloscope
- Instead use a **logic analyzer** to decode bus traffic
- Worth its weight in gold!
  - Analog Discovery and Waveforms
  - Saelae 8-Channel Logic Analyzer
  - Build your own: with Logic Sniffer or related open-source project





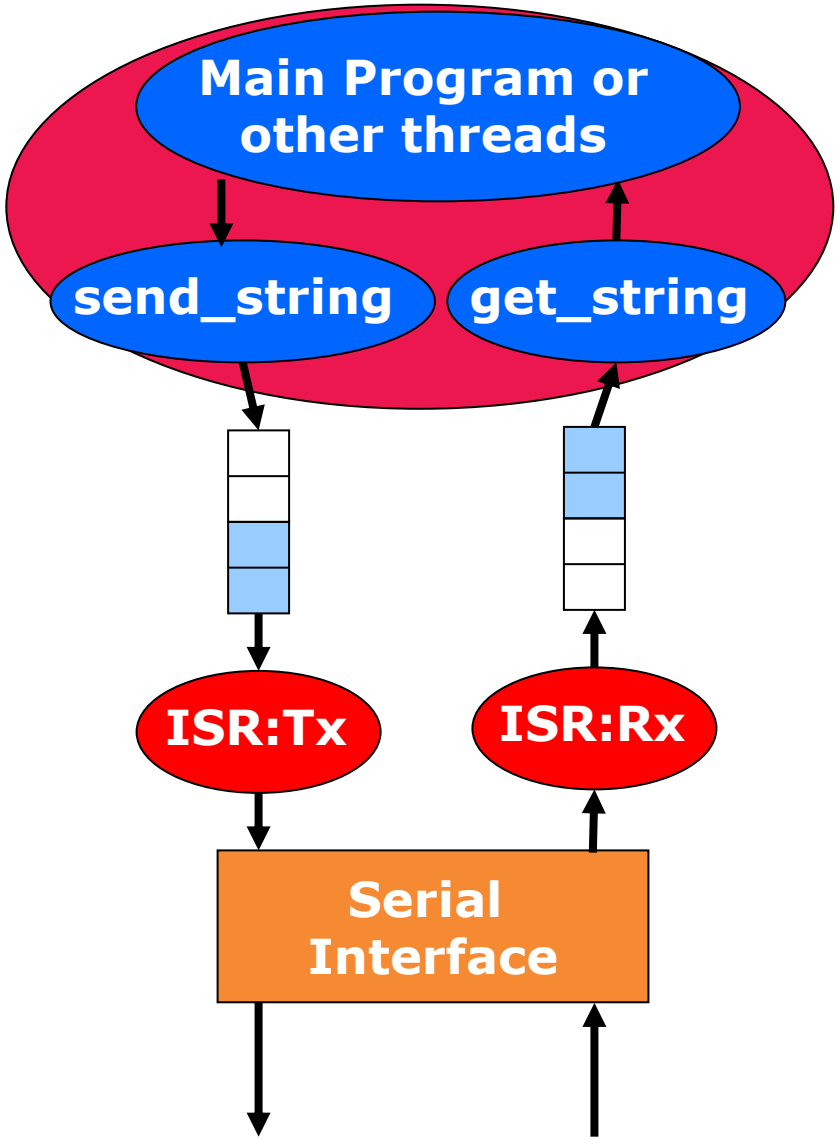
SOFTWARE ARCHITECTURE FOR HANDLING ASYNCHRONOUS COMMUNICATION

Software Structure

- Communication is *asynchronous* to program
  - Don't know what code the program will be executing …
    - when the next item arrives
    - when current outgoing item completes transmission
    - when an error occurs
  - Need to synchronize between program and serial communication interface somehow
- Options
  - Polling
    - Wait until data is available
    - Simple but inefficient of processor time
  - Interrupt
    - CPU interrupts program when data is available
    - Efficient, but more complex

Serial Communications and Interrupts

- Want to provide *multiple* threads of control in the program
  - Main program (and subroutines it calls)
  - ISR(s)
    - Transmit ISR activity – executes when serial interface is ready to send another character
    - Receive ISR activity – executes when serial interface receives a character
    - Error ISR(s) activity – execute if an error occurs
- Need a way of buffering information between threads
  - Solution: circular queue with head and tail pointers
  - One for tx, one for rx



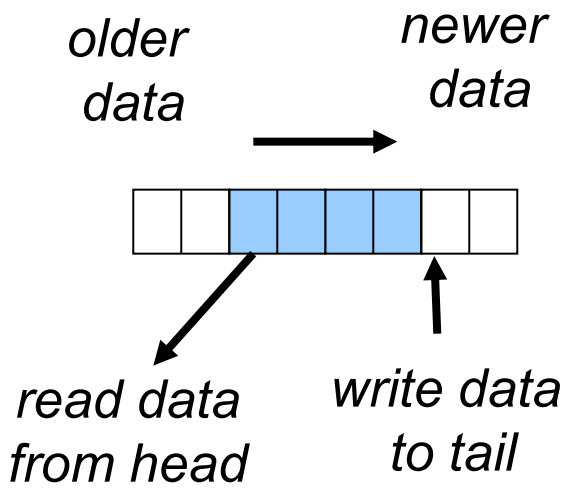
Enabling and Connecting Interrupts to ISRs

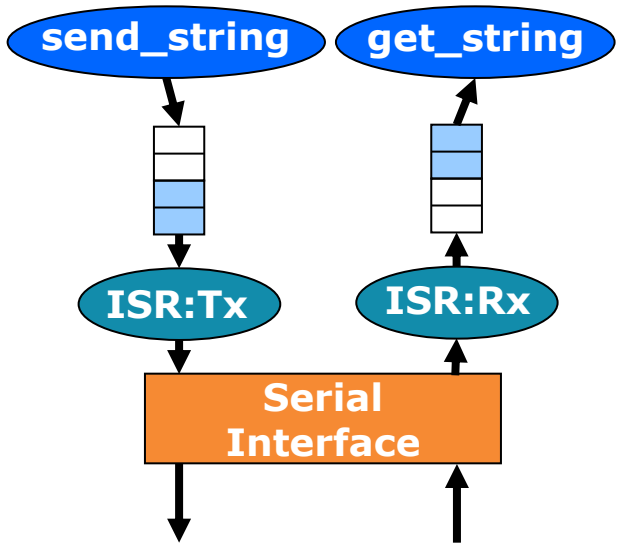
- ARM Cortex-M0+ provides one IRQ for all of a communication interface's events
- Within ISR (IRQ Handler), need to determine what triggered the interrupt, and then service it

```
void UART2_IRQHandler() {
  if (transmitter ready) {
       if (more data to send) {
              get next byte
              send it out transmitter
       }
  }
  if (received data) {
       get byte from receiver
       save it
  }
  if (error occurred) {
       handle error
  }

}
```

Code to Implement Queues

- Enqueue at tail: tail_ptr points to next free entry
- Dequeue from head: head_ptr points to item to remove
- #define the queue size to make it easy to change
- One queue per direction
  - ISR unloads tx_q for transmit
  - ISR loads rx_q for receive
- Other threads (e.g. main) load tx_q and unload rx_q
- Need to wrap pointer at end of buffer to make it circular,
  - Use % (modulus, remainder) operator if queue size is not power of two
  - Use & (bitwise and) if queue size is a power of two
- Queue is empty if size == 0
- Queue is full if size == Q_SIZE





Defining the Queues

```
#define Q_SIZE (32)

typedef struct {
  unsigned char Data[Q_SIZE];
  unsigned int Head; // points to oldest data element
  unsigned int Tail; // points to next free space
  unsigned int Size; // quantity of elements in queue
} Q_T;

Q_T tx_q, rx_q;
```

Initialization and Status Inquiries

```
void Q_Init(Q_T * q) {
  unsigned int i;
  for (i=0; i<Q_SIZE; i++)
    q->Data[i] = 0;  // to simplify our lives when debugging
  q->Head = 0;
  q->Tail = 0;
  q->Size = 0;
}

int Q_Empty(Q_T * q) {
  return q->Size == 0;
}

int Q_Full(Q_T * q) {
  return q->Size == Q_SIZE;
}
```

Enqueue and Dequeue

```
int Q_Enqueue(Q_T * q, unsigned char d) {
  // What if queue is full?
  if (!Q_Full(q)) {
    q->Data[q->Tail++] = d;
    q->Tail %= Q_SIZE;
    q->Size++;
    return 1; // success
  } else
    return 0; // failure
}
unsigned char Q_Dequeue(Q_T * q) {
  // Must check to see if queue is empty before dequeueing
  unsigned char t=0;
  if (!Q_Empty(q)) {
    t = q->Data[q->Head];
    q->Data[q->Head++] = 0; // to simplify debugging
    q->Head %= Q_SIZE;
    q->Size--;
  }
  return t;
}
```

Using the Queues

- Sending data:

```
if (!Queue_Full(…)) {
Queue_Enqueue(…, c)
}
```

- Receiving data:

```
if (!Queue_Empty(…)) {
c=Queue_Dequeue(…)
}
```

SOFTWARE DESIGNS – PARSING MESSAGES

Decoding Messages

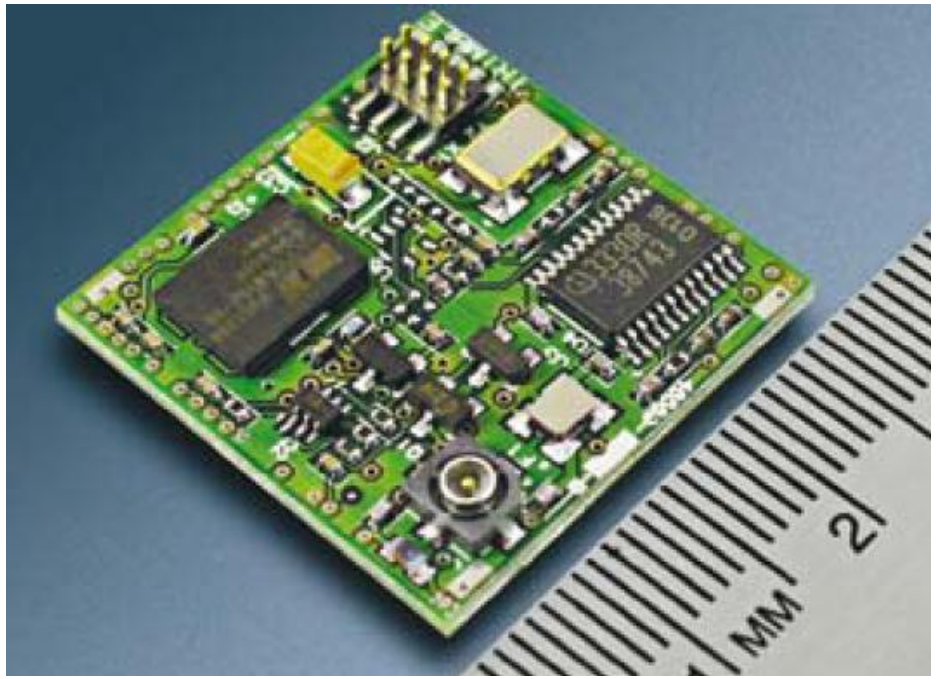
- Two types of messages
- Actual **binary data** sent
  - Detect start of message
  - Identify message type
  - Optional: Confirm integrity with CRC
  - Based on this message type, *copy* binary data from message fields into variables
    - May need to use pointers and casting to get code to translate formats correctly and safely
- **ASCII text** characters representing data sent
  - Detect start of message
  - Identify message type
  - Optional: Confirm integrity with CRC
  - Based on this message type, translate (parse) the data from the ASCII message format into a binary format
  - Copy the binary data into variables

Example UART Application

- Many subsystems connect with the rest of the system using asynchronous serial communications

Table 3.1 Default Protocols and Port Configurations

Port	Input Protocol	Default Setup	Output Language	Default Setup
1	TSIP	Baud Rate: 9600 Data Bits: 8 Parity: Odd Stop Bits: 1 No Flow Control	TSIP	Baud Rate: 9600 Data Bits: 8 Parity: Odd Stop Bits: 1 No Flow Control
2	RTCM	Baud Rate: 4800 Data Bits: 8 Parity: None Stop Bits: 1 No Flow Control	NMEA	Baud Rate: 4800 Data Bits: 8 Parity: None Stop Bits: 1 No Flow Control



- Lassen iQ GPS receiver module from Trimble
  - Two full-duplex asynch. serial connections
  - Three protocols supported
  - Support higher speeds through reconfiguration

Example Binary Serial Data: TSIP

TSIP packet structure is the same for both commands and reports. The packet format is:

<DLE> <id> <data string bytes> <DLE> <ETX>

Where:

- <DLE> is the byte 0x10
- <ETX> is the byte 0x03
- <id> is a packet identifier byte, which can have any value excepting <ETX> and <DLE>.

```
switch (id) {
case 0x84:
   lat = *((double *) (&msg[0]));
   lon = *((double *) (&msg[8]));
   alt = *((double *) (&msg[16]));
   clb = *((double *) (&msg[24]));
   tof = *((float  *) (&msg[32]));
   break;
case 0x4A: …

default:
   break;
}
```

Table A.52 Report Packet 0x84 Data Formats

Byte	Item	Type	Units
0-7	latitude	Double	radians; + for north, - for south
8-15	longitude	Double	radians; + for east, - for west
16-23	altitude	Double	meters
24-31	clock bias	Double	meters
32-35	time-of-fix	Single	seconds

Output ID	Packet Description
0x41	GPS time
0x42	single-precision XYZ position
0x43	velocity fix (XYZ ECEF)
0x45	software version information
0x46	health of Receiver
0x47	signal level for all satellites
0x4A	single-precision LLA position
0x4B	machine code/status
0x4D	oscillator offset
0x4E	response to set GPS time
0x55	I/O options
0x56	velocity fix (ENU)

Example ASCII Serial Data: NMEA-0183

```
$IDMSG,D1,D2,D3,D4,.....,Dn*CS [CR] [LF]
```

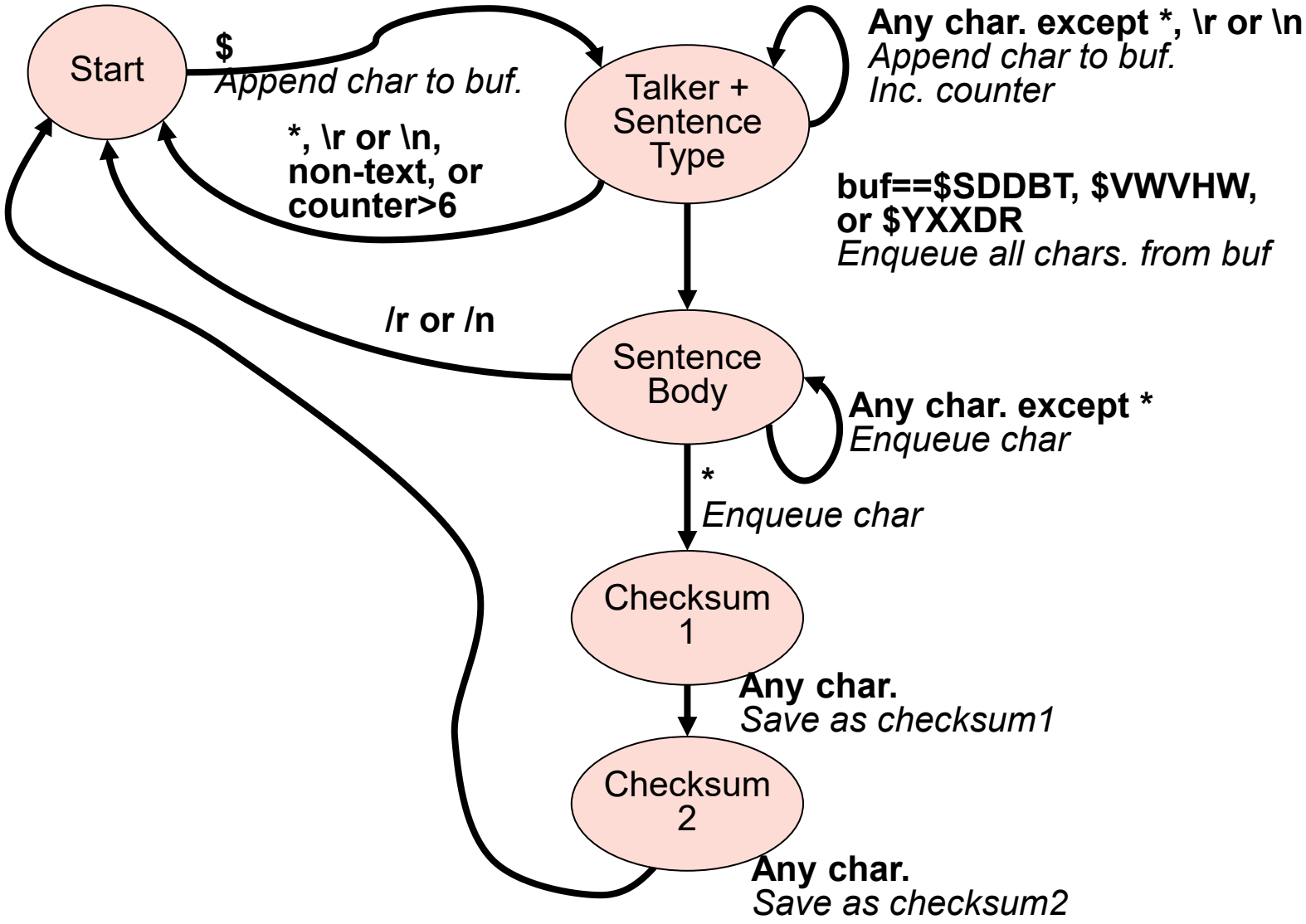
"$"	The "$" signifies the start of a message.
ID	The talker identification is a two letter mnemonic which describes the source of the navigation information. The GP identification signifies a GPS source.
MSG	The message identification is a three letter mnemonic which describes the message content and the number and order of the data fields.
","	Commas serve as delimiters for the data fields.
Dn	Each message contains multiple data fields (Dn) which are delimited by commas.
"*"	The asterisk serves as a checksum delimiter.
CS	The checksum field contains two ASCII characters which indicate the hexadecimal value of the checksum.
[CR][LF]	The carriage return [CR] and line feed [LF] combination terminate the message.

```
$GPRMC,hhmmss.ss,A,llll.ll,a,yyyyy.yy,a,
x.x,x.x,xxxxxx,x.x,a,i*hh<CR><LF>
```

Table E.8 RMC - Recommended Minimum Specific GPS / Transit Data Message Parameters

Field #	Description
1	UTC of Position Fix (when UTC offset has been decoded by the receiver).
2	Status: A = Valid, V = navigation receiver warning
3,4	Latitude, N (North) or S (South).
5,6	Longitude, E (East) or W (West).
7	Speed over the ground (SOG) in knots
8	Track made good in degrees true.
9	Date: dd/mm/yy
10,11	Magnetic variation in degrees, E = East / W= West
12	Position System Mode Indicator; A=Autonomous, D=Differential, E=Estimated (Dead Reckoning), M=Manual Input, S=Simulation Mode, N=Data Not Valid
hh	Checksum (Mandatory for RMC)

State Machine for Parsing NMEA-0183

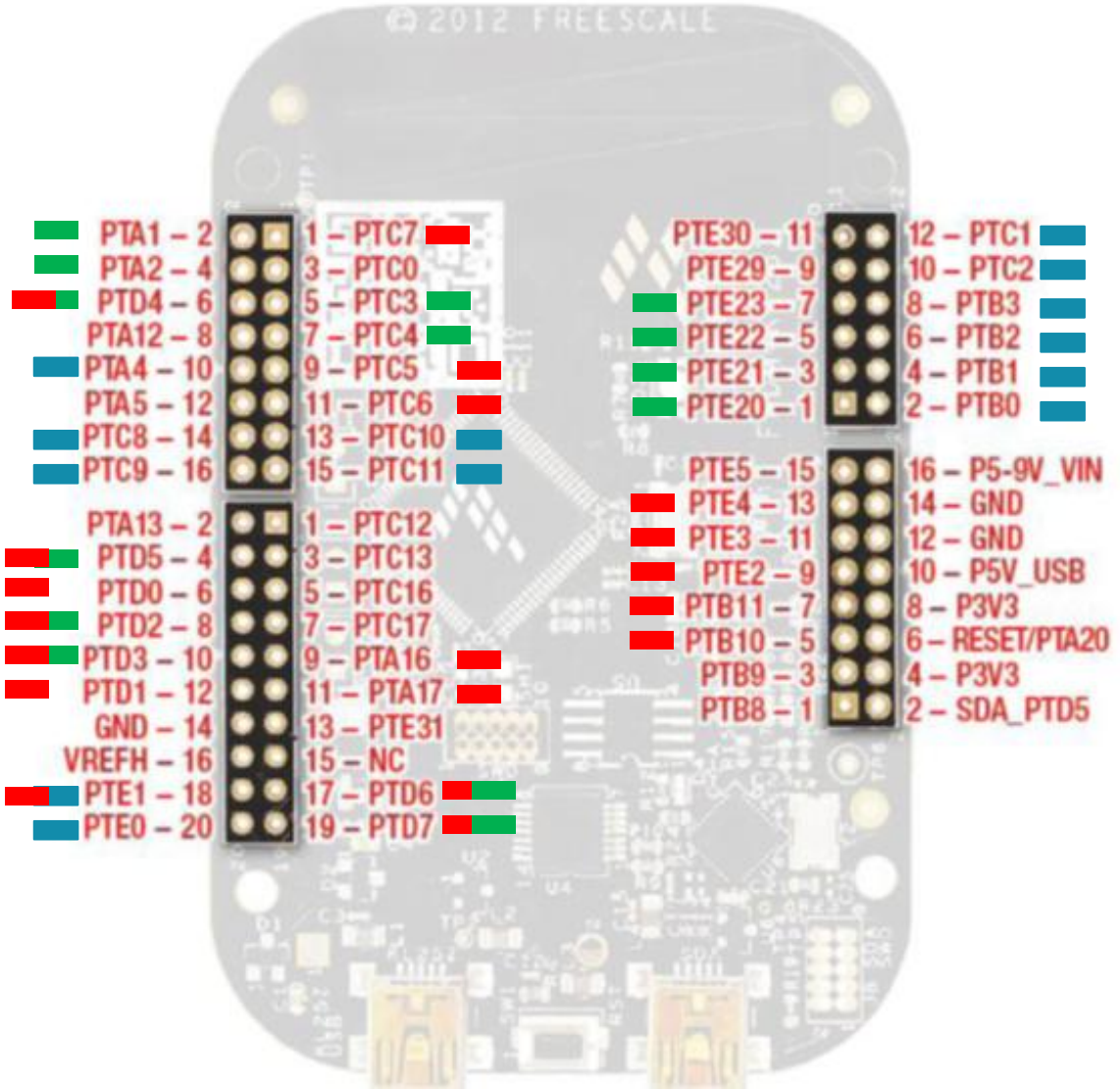


Parsing

```
switch (parser_state) {
case TALKER_SENTENCE_TYPE:
  switch (msg[i]) {
        '*':
        '\r':
        '\n':
                parser_state = START;
                break;
        default:
                if (Is_Not_Character(msg[i]) || n>6) {
                        parser_state = START;
                } else {
                        buf[n++] = msg[i];
                }
        break;
  }
  if ((n==6) & … ){
        parser_state = SENTENCE_BODY;
  }
  break;
case SENTENCE_BODY:
  break;
```

KL25Z AND FREEDOM SPECIFICS

Freedom KL25Z Serial I/O



UART

SPI

I^2C

KL25Z Clock Gating for Serial Comm.

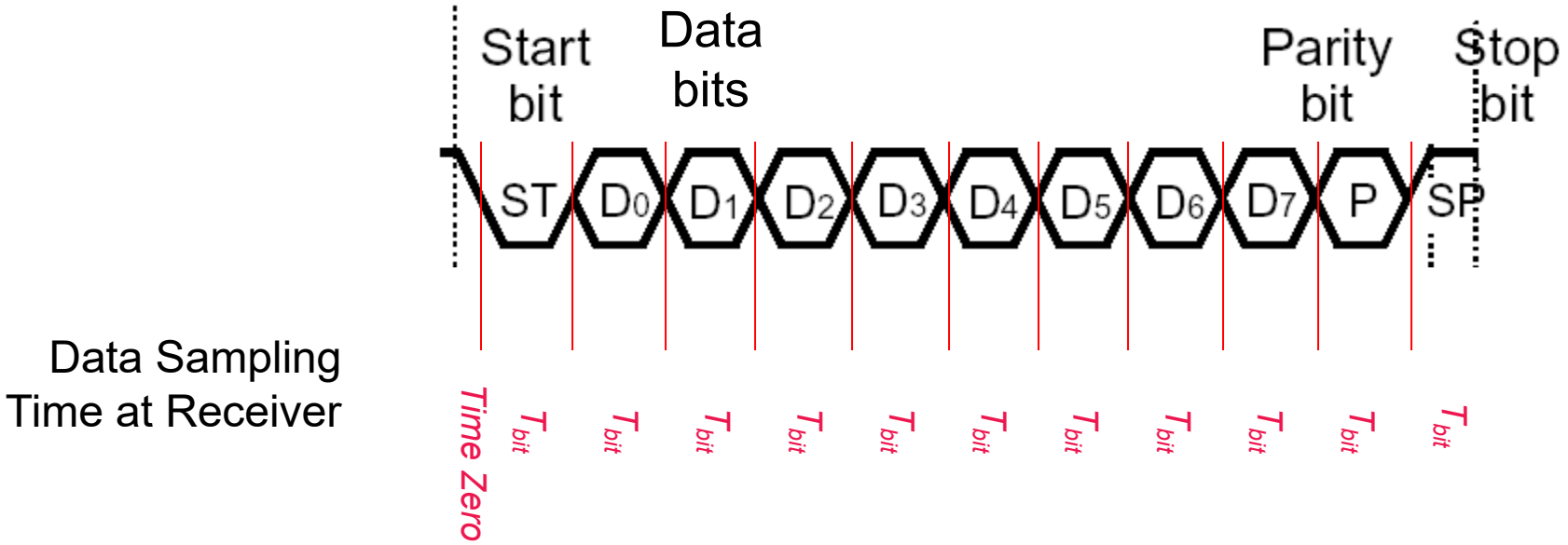
Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	1				0				SPI1	SPI0	0		CMP	USBOTG	0	
W																
Reset	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0		0	UART2	UART1	UART0	0		I2C1	I2C0	1		0			
W																
Reset	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0

- Set corresponding bit(s) in SIM_SCGC4 Register

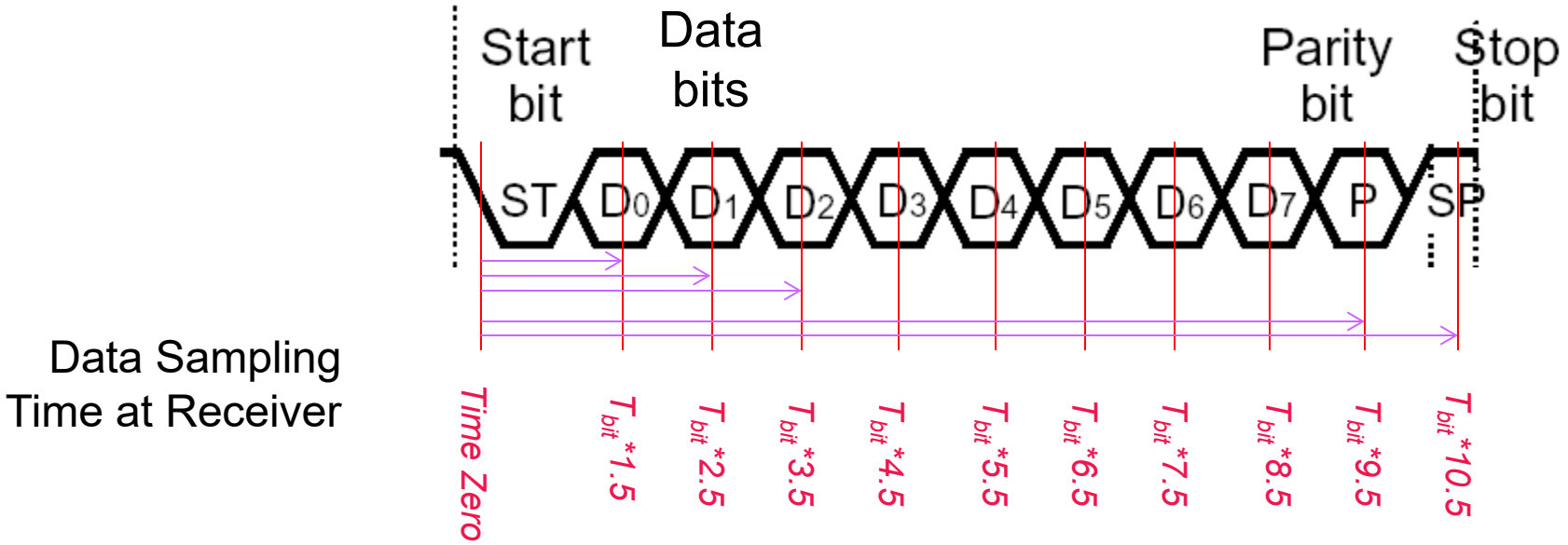
ASYNCHRONOUS SERIAL (UART) COMMUNICATIONS

Transmitter Basics



- If no data to send, keep sending 1 (stop bit) – *idle line*
- When there is a data word to send
  - Send a 0 (start bit) to indicate the start of a word
  - Send each data bit in the word (use a shift register for the *transmit buffer*)
  - Send a 1 (stop bit) to indicate the end of the word

Receiver Basics



- Wait for a falling edge (beginning of a Start bit)
  - Then wait ½ bit time
  - Do the following for as many data bits in the word
    - Wait 1 bit time
    - Read the data bit and shift it into a *receive buffer* (shift register)
  - Wait 1 bit time
  - Read the bit
    - if 1 (Stop bit), then OK
    - if 0, there's a problem!

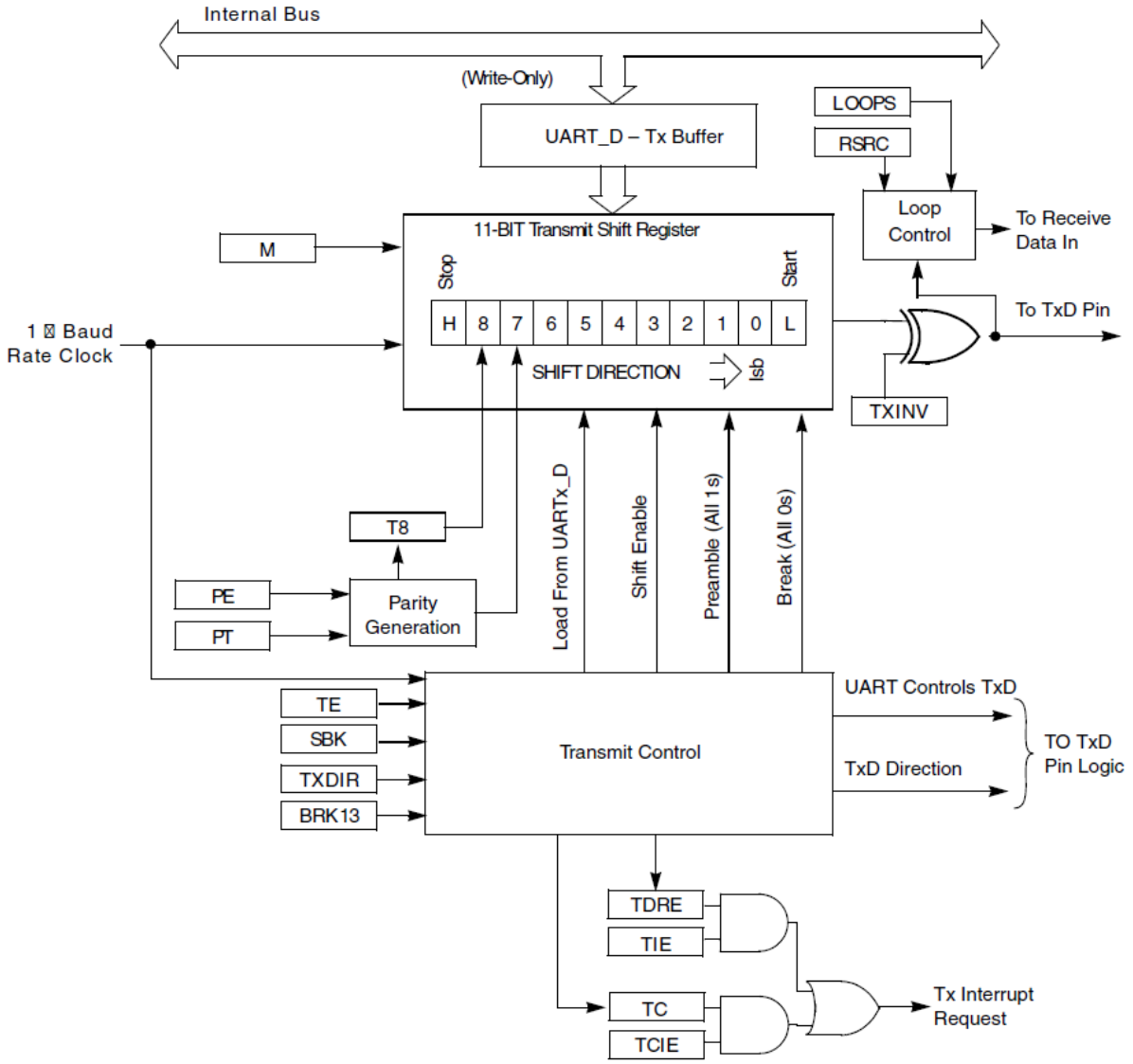
For this to work…

- Transmitter and receiver must agree on several things (protocol)
  - Order of data bits
  - Number of data bits
  - What a start bit is (1 or 0)
  - What a stop bit is (1 or 0)
  - How long a bit lasts
    - Transmitter and receiver clocks must be reasonably close in frequency, since the only timing reference is the start of the start bit

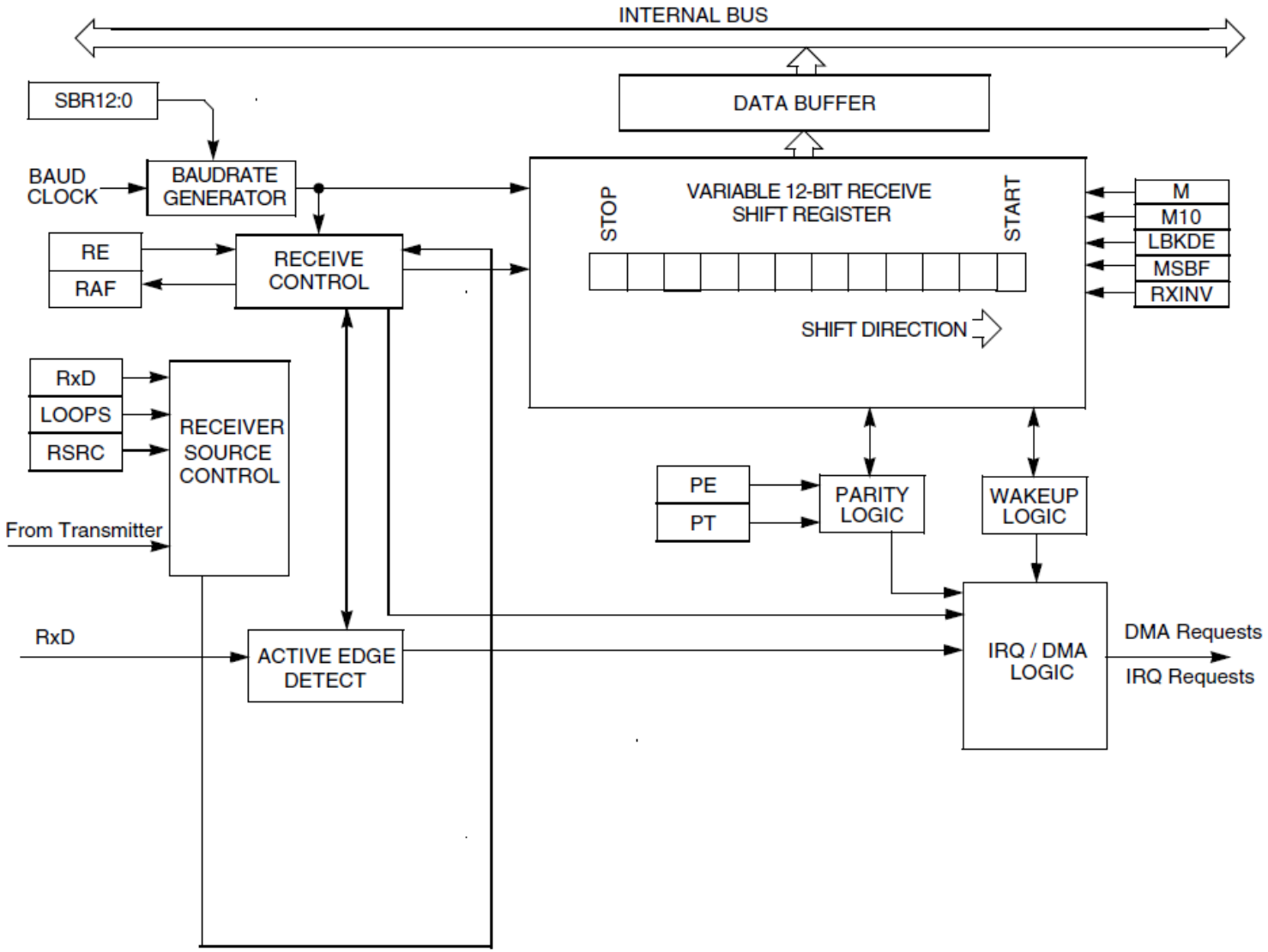
KL25 UARTs

- UART: Universal (configurable) Asynchronous Receiver/Transmitter
- UART0
  - Low Power
  - Can oversample from 4x to 32x
  - Is used by debugger MCU on Freedom KL25Z, so not available
- UART1, UART2
  - More basic, fewer features, easier to program

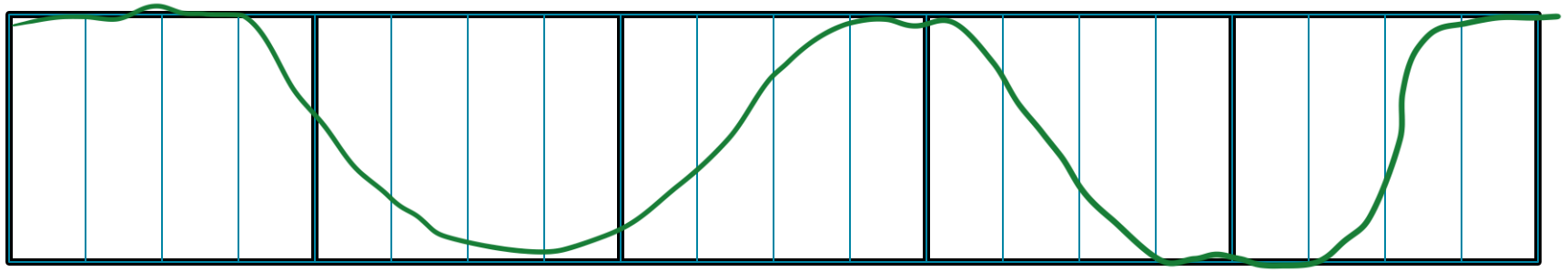
UART Transmitter



UART Receiver

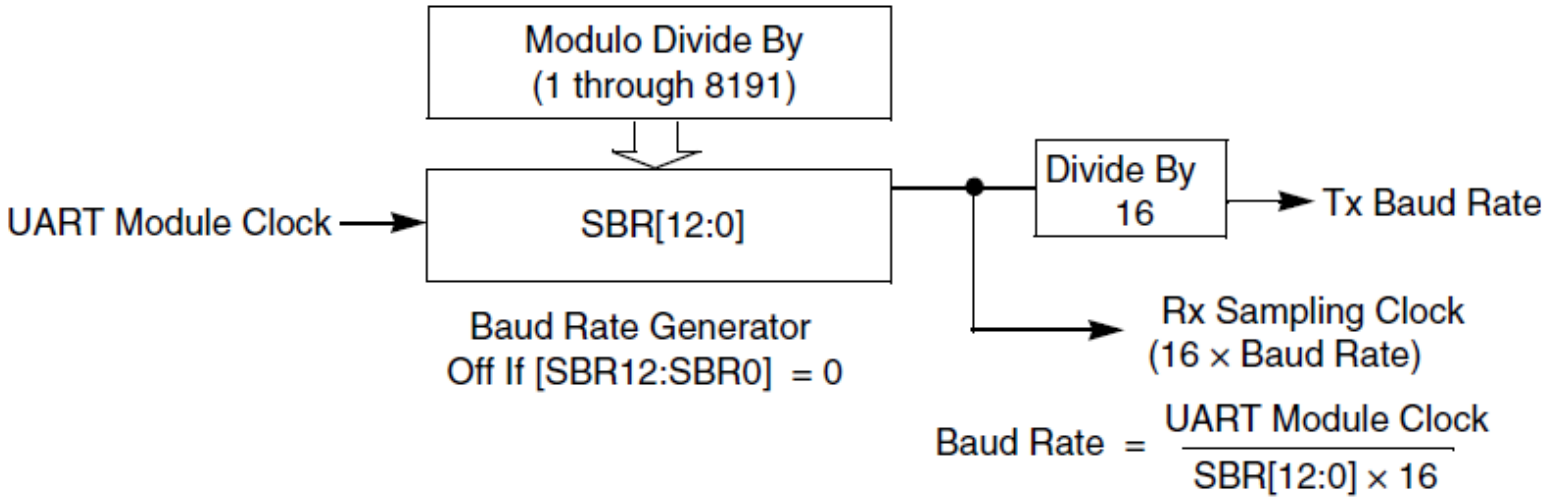


Input Data Oversampling



- When receiving, UART *oversamples* incoming data line
  - Extra samples allow voting, improving noise immunity
  - Better synchronization to incoming data, improving noise immunity
- UART0 provides configurable oversampling from 4x to 32x
  - Put desired oversampling factor minus one into UART0 Control Register 4, OSR bits.
- UART1, UART2 have fixed 16x oversampling

Baud Rate Generator



- Need to divide module clock frequency down to desired baud rate * oversampling factor
- Example
  - 24 MHz -> 4800 baud with 16x oversampling
  - Division factor = 24E6/(4800*16) = 312.5. Must round to closest integer value (312 or 313), will have a slight frequency error.

Using the UART

- When can we transmit?
  - Transmit buffer must be empty
  - Can poll UARTx->S1 TDRE flag
  - Or we can use an interrupt, in which case we will need to queue up data
- Put data to be sent into UARTx_D (UARTx->D)

- When can we receive a byte?
  - Receive buffer must be full
  - Can poll UARTx->S1 RDRF flag
  - Or we can use an interrupt, and again we will need to queue the data
- Get data from UARTx_D (UARTx->D)

UART Control Register 1 (UART0_C1)

Bit	7	6	5	4	3	2	1	0
Read Write	LOOPS	DOZEEN	RSRC	M	WAKE	ILT	PE	PT
Reset	0	0	0	0	0	0	0	0

- LOOPS: Enables loopback/single-pin (TX/RX) mode
- DOZEEN: Doze enable – disable UART in sleep mode
- RSRC: Selects between loopback and single-pin mode
- M: Select 9-bit data mode (instead of 8-bit data)
- WAKE: Wakeup method
- ILT: Idle line type
- PE: Parity enabled with 1
- PT: Odd parity with 1, even parity with 0

UART Control Register 2 (UART0_C2)

Bit	7	6	5	4	3	2	1	0
Read Write	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
Reset	0	0	0	0	0	0	0	0

- Interrupt Enables
  - TIE: Interrupt when Transmit Data Register is empty
  - TCIE: Interrupt when transmission completes
  - RIE: Interrupt when receiver has data ready
- Module Enables
  - TE: Transmitter enable
  - RE: Receiver enable
- Other
  - RWU: Put receiver in standby mode, will wake up when condition occurs
  - SBK: Send a break character (all zeroes)

UART Status Register 1 (UART_S1)

Bit	7	6	5	4	3	2	1	0
Read	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF
Write				w1c	w1c	w1c	w1c	w1c
Reset	1	1	0	0	0	0	0	0

- TDRE: Transmit data register empty, can write more data to data register
- TC: Transmission complete.
- RDRF: Receiver data register full, can read data from data register
- IDLE: UART receive line has been idle for one full character time
- OR: Receive overrun. Received data has overwritten previous data in receive buffer
- NF: Noise flag. Receiver data bit samples don’t agree.
- FE: Framing error. Received 0 for a stop bit, expected 1.
- PF: Parity error. Incorrect parity received.

UART Status Register 2 (UARTx_S2)

Bit	7	6	5	4	3	2	1	0
Read	LBKDIF	RXEDGIF	MSBF	RXINV	RWUID	BRK13	LBKDE	RAF
Write								
Reset	0	0	0	0	0	0	0	0

- LBDIF: LIN break detect interrupt flag
- RXEDGIF: Active edge on receive pin detected
- MSBF: Send MSB first. Should be 0 for RS232
- RXINV: Invert received signals (data, start, stop, etc.)
- RWUID: Set idle bit upon wakeup?
- BRK13: Set break character to 13 bits long (not 10)
- LBKDE: LIN break character time.
- RAF: Receiver is actively receiving data (not idle line)

Software for Polled Serial Comm.

```
void Init_UART2(uint32_t baud_rate) {
        uint32_t divisor;
        // enable clock to UART and Port A
        SIM->SCGC4 |= SIM_SCGC4_UART2_MASK;
        SIM->SCGC5 |= SIM_SCGC5_PORTE_MASK;

        // connect UART to pins for PTE22, PTE23
        PORTE->PCR[22] = PORT_PCR_MUX(4);
        PORTE->PCR[23] = PORT_PCR_MUX(4);
        // ensure tx and rx are disabled before configuration
        UART2->C2 &=  ~(UARTLP_C2_TE_MASK | UARTLP_C2_RE_MASK);

        // Set baud rate to 4800 baud
        divisor = BUS_CLOCK/(baud_rate*16);
        UART2->BDH = UART_BDH_SBR(divisor>>8);
        UART2->BDL = UART_BDL_SBR(divisor);

        // No parity, 8 bits, two stop bits, other settings;
        UART2->C1 = UART2->S2 = UART2->C3 = 0;

        // Enable transmitter and receiver
        UART2->C2 = UART_C2_TE_MASK | UART_C2_RE_MASK;
}
```

Polled Serial Transmitter Code

```
void UART2_Transmit_Poll(uint8_t data) {
      // wait until transmit data register is empty
      while (!(UART2->S1 & UART_S1_TDRE_MASK))
            ;
      UART2->D = data;
}
void main(void) {
      char c;
      // Initialization goes here
      while (1) {
            for (c='a'; c<='z'; c++) {
                  UART2_Transmit_Poll(c);
}
```

Polled Serial Receiver Code with Echo

```
uint8_t UART2_Receive_Poll(void) {
      // wait until receive data register is full
      while (!(UART2->S1 & UART_S1_RDRF_MASK))
            ;
      return UART2->D;
}
void main(void) {
      char c;
      // Initialization goes here
      while (1) {
            c = UART2_Receive_Poll();
            UART2_Transmit_Poll(c);
}
```

Software for Interrupt-Driven Serial Comm.

- Use interrupts
- First, initialize peripheral to generate interrupts
- Second, create single ISR with three sections corresponding to cause of interrupt
  - Transmitter
  - Receiver
  - Error

Peripheral Initialization

```
void Init_UART2(uint32_t baud_rate) {
        …
        NVIC_SetPriority(UART2_IRQn, 2);
        NVIC_ClearPendingIRQ(UART2_IRQn);
        NVIC_EnableIRQ(UART2_IRQn);

        UART2->C2 |= UART_C2_TIE_MASK |
                             UART_C2_RIE_MASK;
        UART2->C2 |= UART_C2_RIE_MASK;
        Q_Init(&TxQ);
        Q_Init(&RxQ);
}
```

Interrupt Handler: Transmitter

```
void UART2_IRQHandler(void) {
      NVIC_ClearPendingIRQ(UART2_IRQn);
      if (UART2->S1 & UART_S1_TDRE_MASK) {
            // can send another character
            if (!Q_Empty(&TxQ)) {
                  UART2->D = Q_Dequeue(&TxQ);
            } else {
                  // queue is empty so disable tx
                  UART2->C2 &= ~UART_C2_TIE_MASK;
            }
      }
      …
```

Interrupt Handler: Receiver

```
void UART2_IRQHandler(void) {
        …
        if (UART2->S1 & UART_S1_RDRF_MASK) {
                // received a character
                if (!Q_Full(&RxQ)) {
                        Q_Enqueue(&RxQ, UART2->D);
                } else {
                        // error - queue full.
                        while (1)
                                ;
                }
        }
```

Interrupt Handler: Error Cases

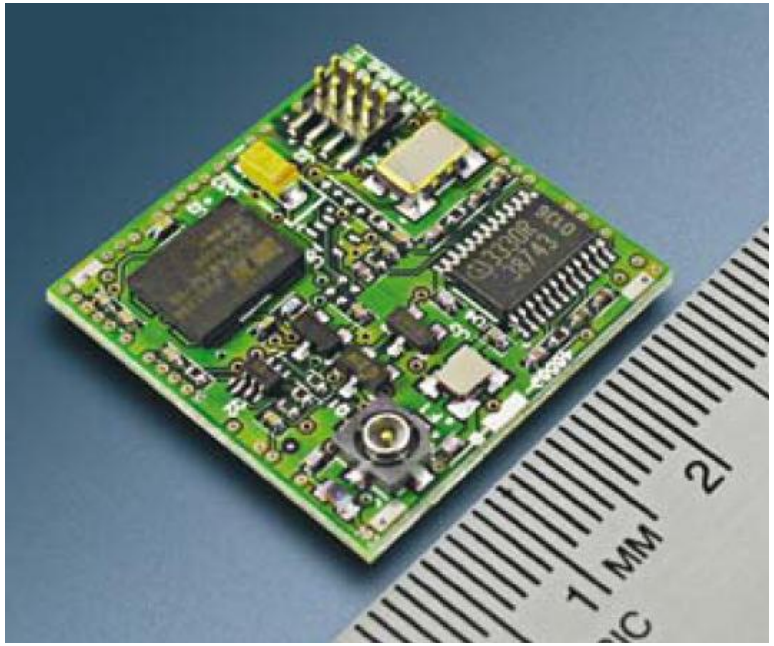
```
void UART2_IRQHandler(void) {
      …
      if (UART2->S1 & (UART_S1_OR_MASK |
                       UART_S1_NF_MASK |
                       UART_S1_FE_MASK |
                       UART_S1_PF_MASK)) {
                  // handle the error

                  // clear the flag

            }
}
```

Example UART Application

- Many subsystems connect with the rest of the system using asynchronous serial communications

Table 3.1 Default Protocols and Port Configurations

Port	Input Protocol	Default Setup	Output Language	Default Setup
1	TSIP	Baud Rate: 9600 Data Bits: 8 Parity: Odd Stop Bits: 1 No Flow Control	TSIP	Baud Rate: 9600 Data Bits: 8 Parity: Odd Stop Bits: 1 No Flow Control
2	RTCM	Baud Rate: 4800 Data Bits: 8 Parity: None Stop Bits: 1 No Flow Control	NMEA	Baud Rate: 4800 Data Bits: 8 Parity: None Stop Bits: 1 No Flow Control

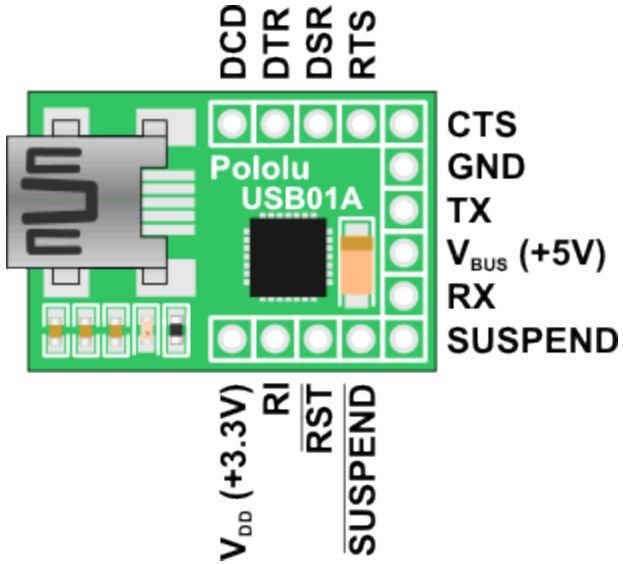


- Lassen iQ GPS receiver module from Trimble
  - Two full-duplex asynch. serial connections
  - Three protocols supported
  - Support higher speeds through reconfiguration

USB to UART Interface

- PCs haven't had external asynchronous serial interfaces for a while, so how do we communicate with a UART?

- USB to UART interface
  - USB connection to PC
  - Logic level (0-3.3V) to microcontroller's UART (not RS232 voltage levels)



- USB01A USB to serial adaptor
  - http://www.pololu.com/catalog/product/391
  - Can also supply 5 V, 3.3 V from USB

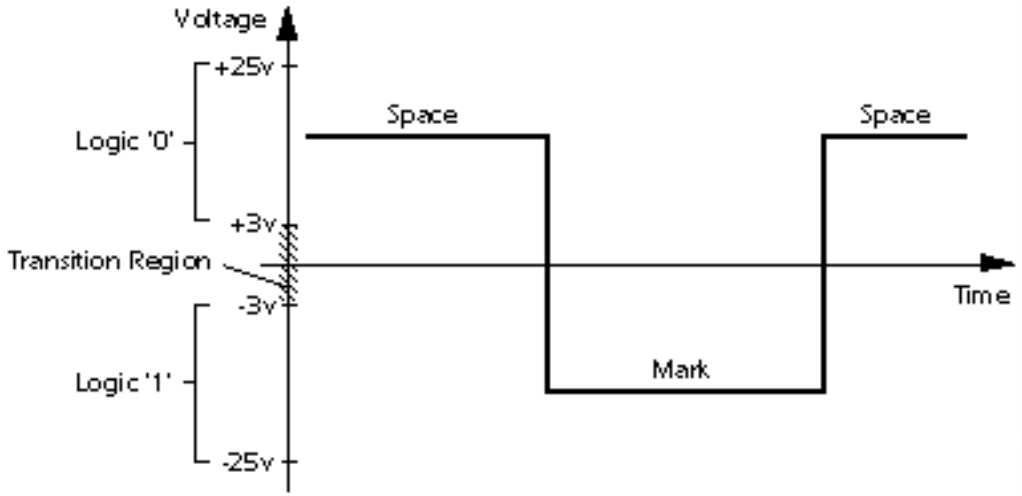
Building on Asynchronous Comm.

- Asynchronous communication is useful but runs into some problems when applying it to some applications

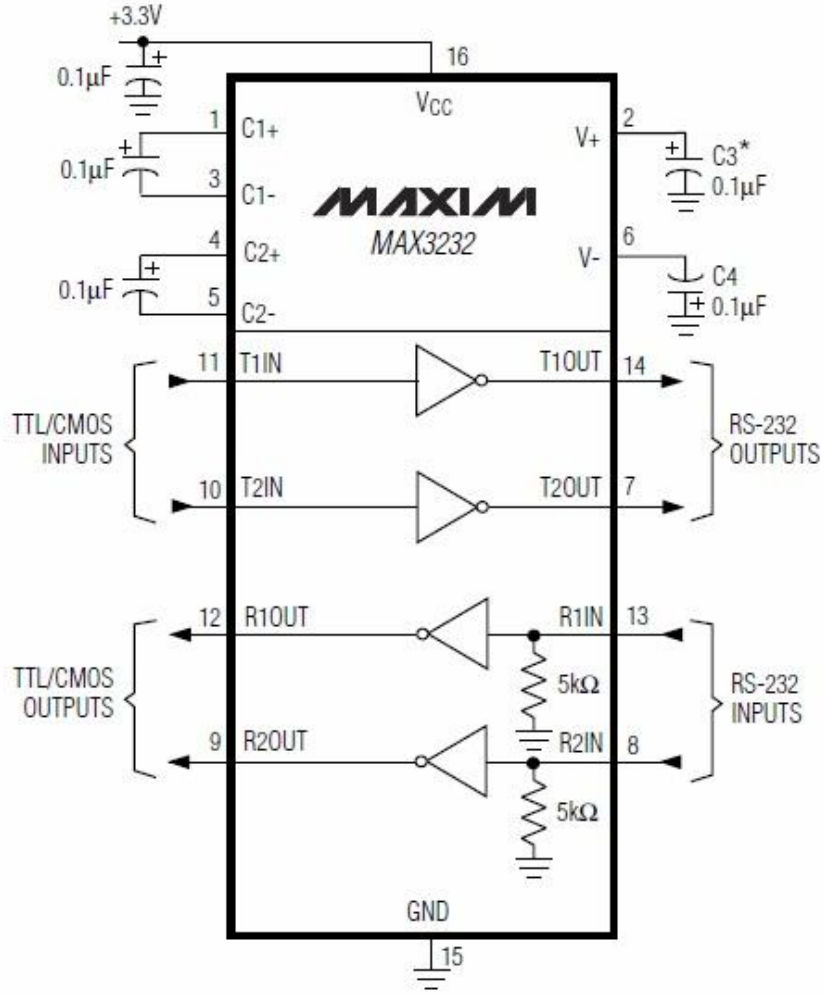
- Problem #1
  - Logic-level signals (0 to 1.65 V, 1.65 V to 3.3 V) are sensitive to noise and signal degradation

- Problem #2
  - Point-to-point topology does not support a large number of nodes well
    - Need a dedicated wire to send information from one device to another
    - Need a UART channel for each device the MCU needs to talk to
    - Single transmitter, single receiver per data wire

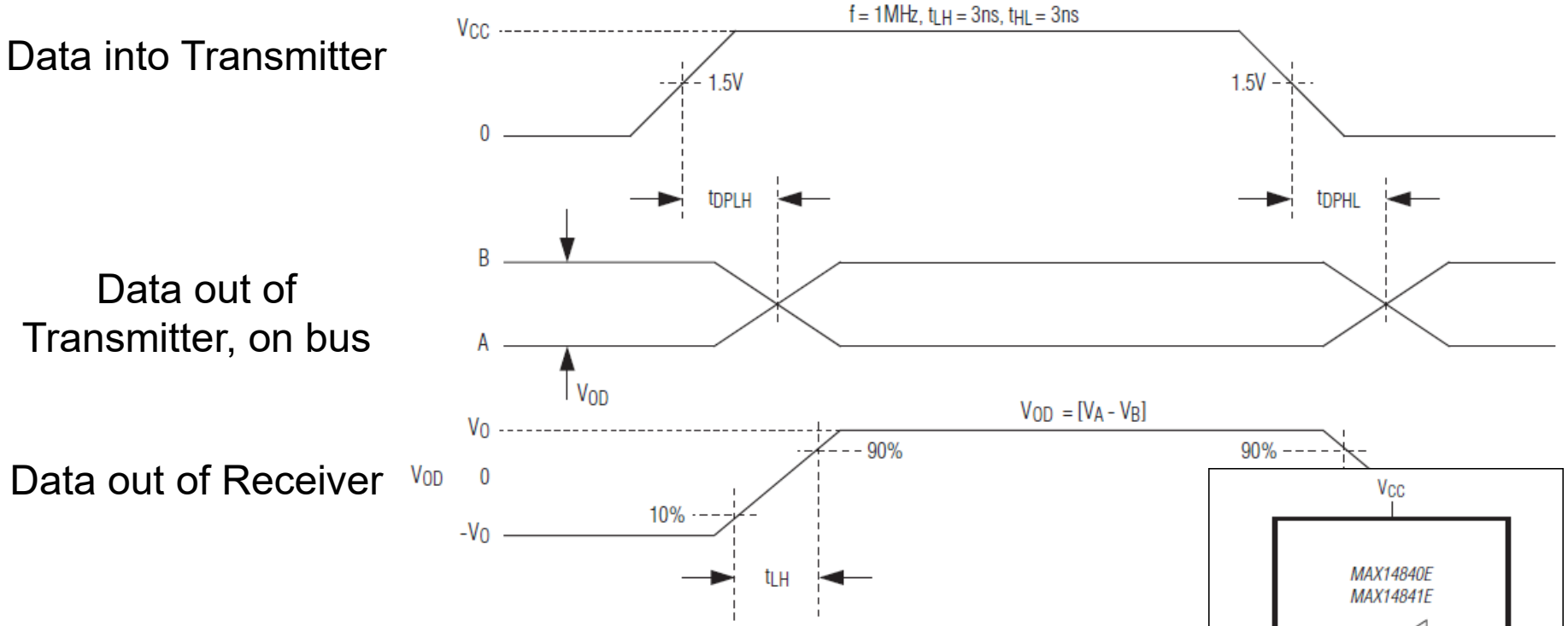
Solution to Noise: Higher Voltages

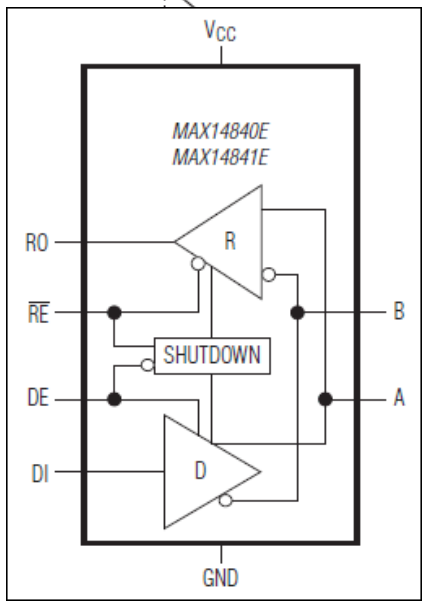


- Use higher voltages to improve noise margin +3 to +15 V, -3 to -15 V
- Example IC (Maxim MAX3232) uses charge pumps to generate higher voltages from 3.3\ supply rail



Solution to Noise: Differential Signaling





- Use differential signaling
  - Send two signals: Buffered data (A), buffered complement of data (B)
  - Receiver compares the two signals to determine if data is a one (A > B) or a zero (B > A)

Solutions to Poor Scaling

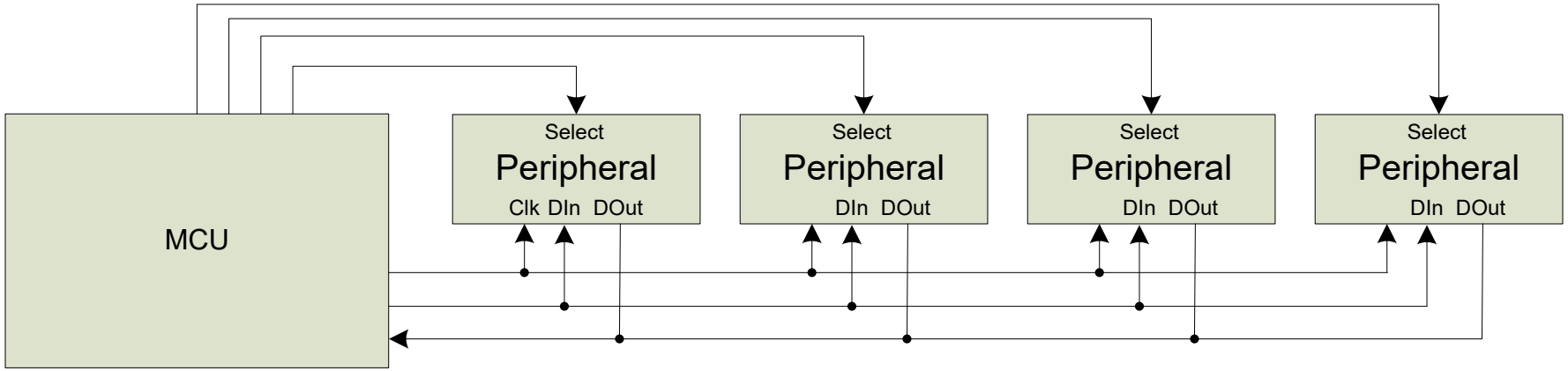
- Approaches
  - Allow one transmitter to drive multiple receivers (multi-drop)
  - Connect all transmitters and all receivers to same data line (multi-point network). Need to add a medium access control technique so all nodes can share the wire
- Example Protocols
  - RS-232: higher voltages, point-to-point
  - RS-422: higher voltages, differential data transmission, multi-drop
  - RS-485: higher voltages, multi-point

Example Protocols

- RS-232: higher voltages, point-to-point
- RS-422: higher voltages, differential data transmission, multi-drop
- RS-485: higher voltages, multi-point

SPI COMMUNICATIONS

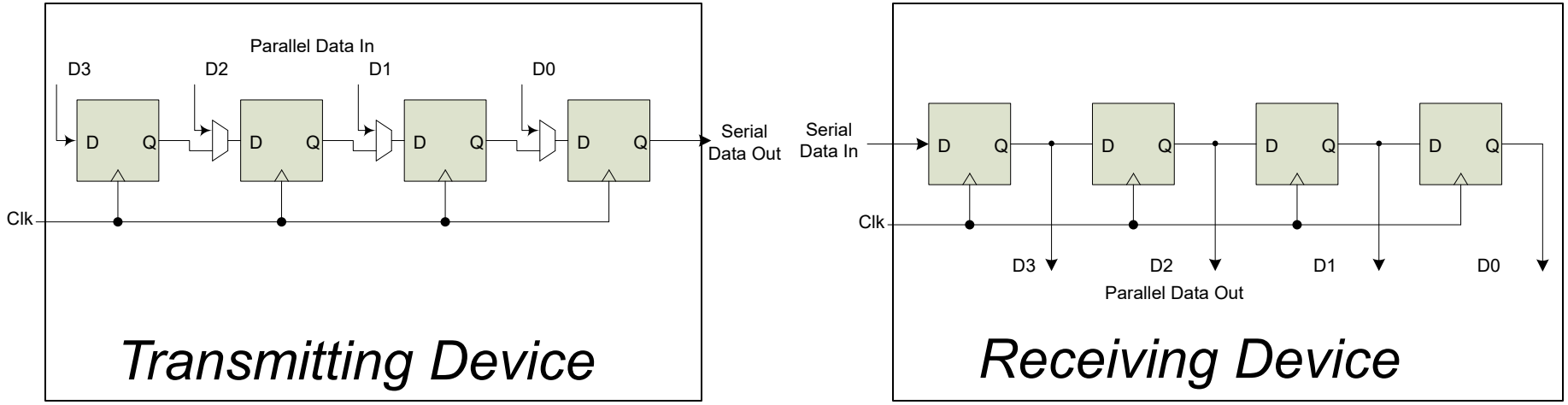
Hardware Architecture

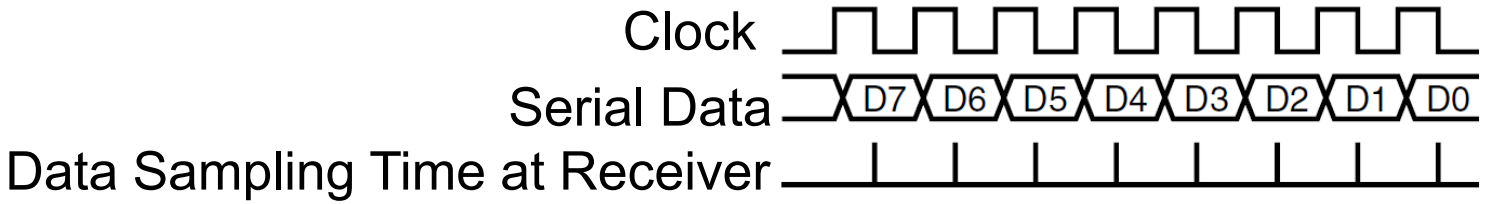


- All chips share bus signals
  - Clock SCK
  - Data lines MOSI (master out, slave in) and MISO (master in, slave out)

- Each peripheral has its own chip select line (CS)
  - Master (MCU) asserts the CS line of only the peripheral it's communicating with

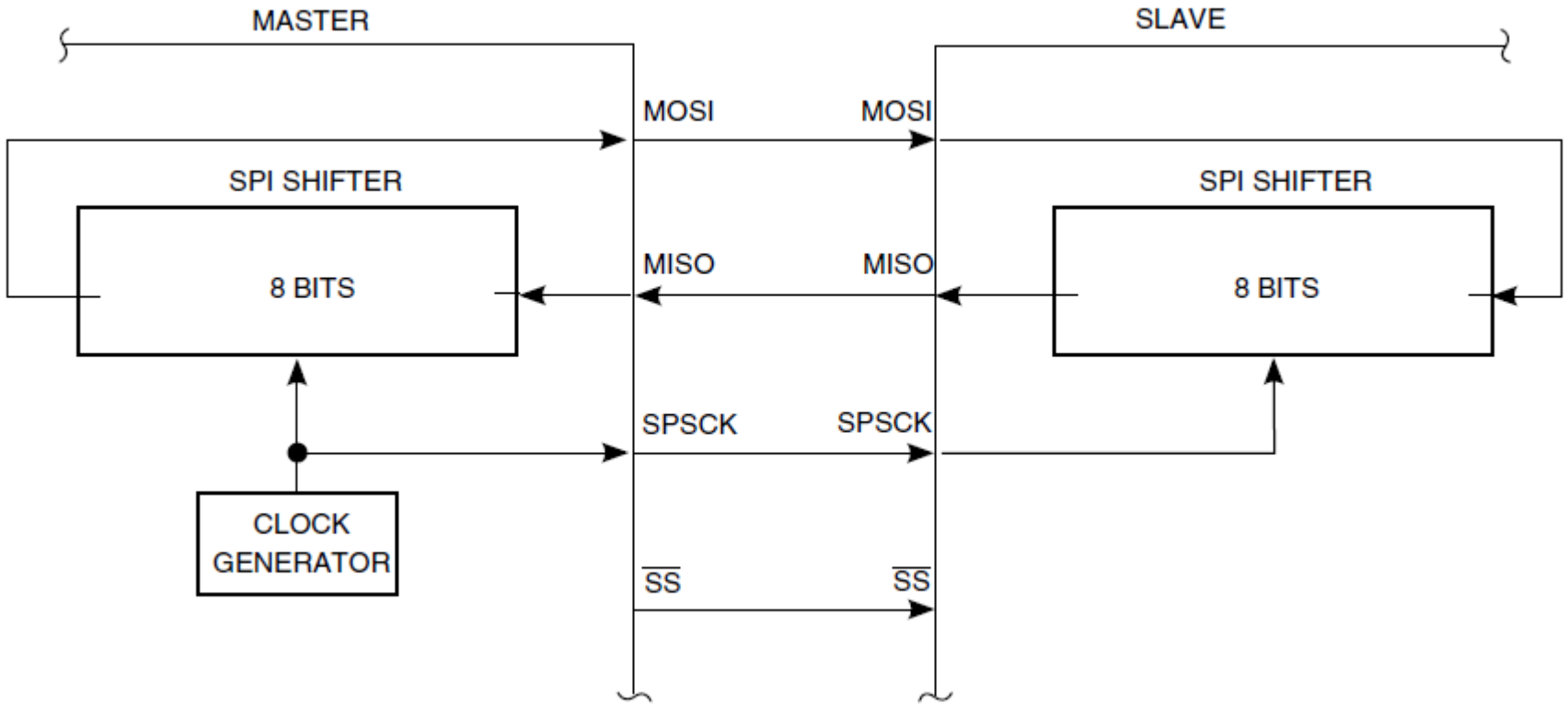
Serial Data Transmission





- Use shift registers and a clock signal to convert between serial and parallel formats
- Synchronous: an explicit clock signal is along with the data signal

SPI Signal Connection Overview



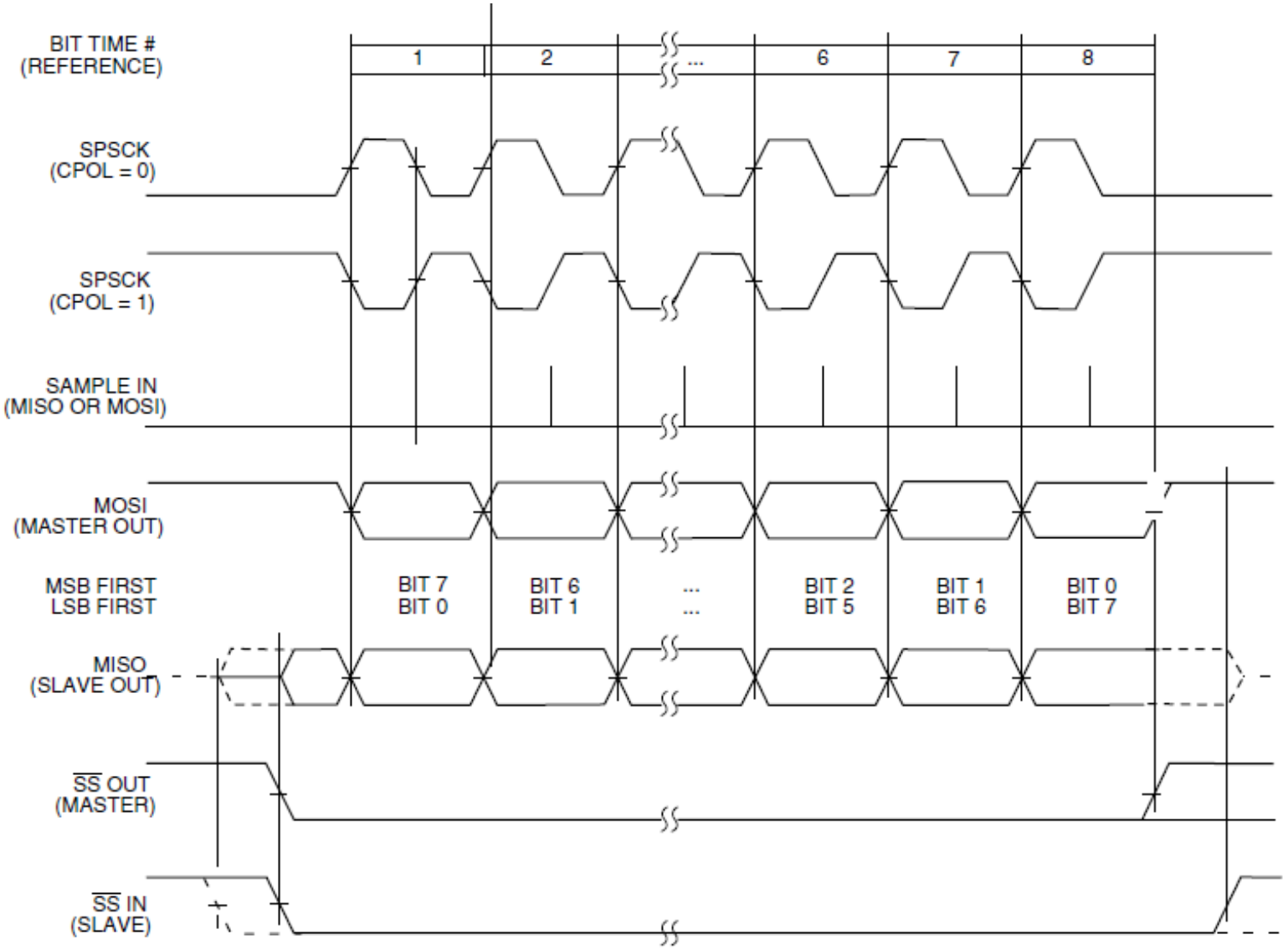
- SPI Communication consists of a series of data swaps between the Master and the Slave
  - As the master shifts out its transmit byte, it is also shifting in the received byte from the Slave

SPI Control Register 1 (SPIx_C1)

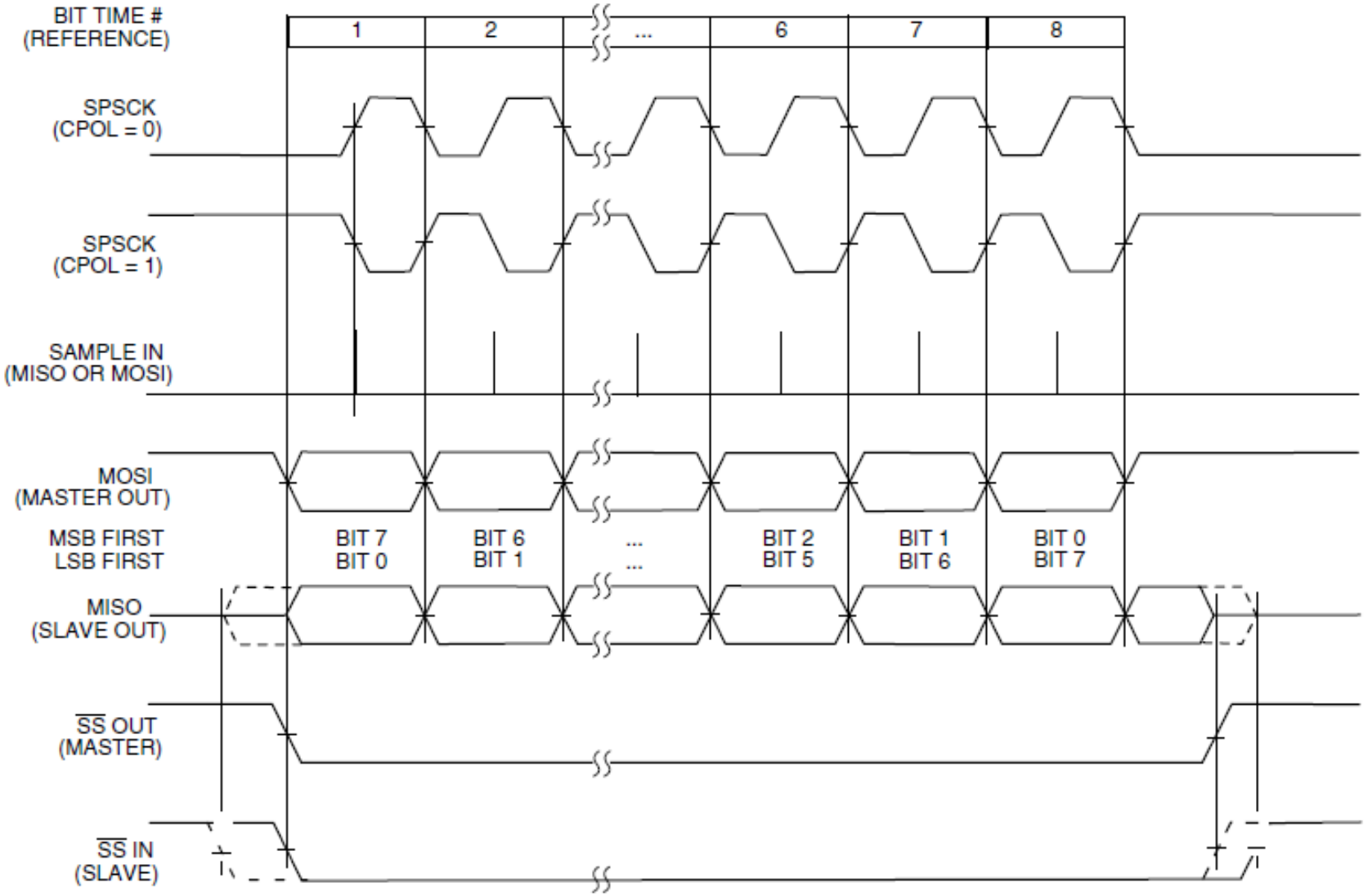
Bit	7	6	5	4	3	2	1	0
Read Write	SPIE	SPE	SPTIE	MSTR	CPOL	CPHA	SSOE	LSBFE
Reset	0	0	0	0	0	1	0	0

- SPIE: SPI interrupt enable for receive buffer full and mode fault
- SPE: SPI system enable
- SPTIE: SPI interrupt enable for transmit buffer empty
- MSTR: select master mode
- CPOL: Clock polarity
- CPHA: Clock phase
- SSOE: Slave select output enable

Clock and Phase Settings: CPHA = 1



Clock and Phase Settings: CPHA = 0



SPI Control Register 2 (SPIx_C2)

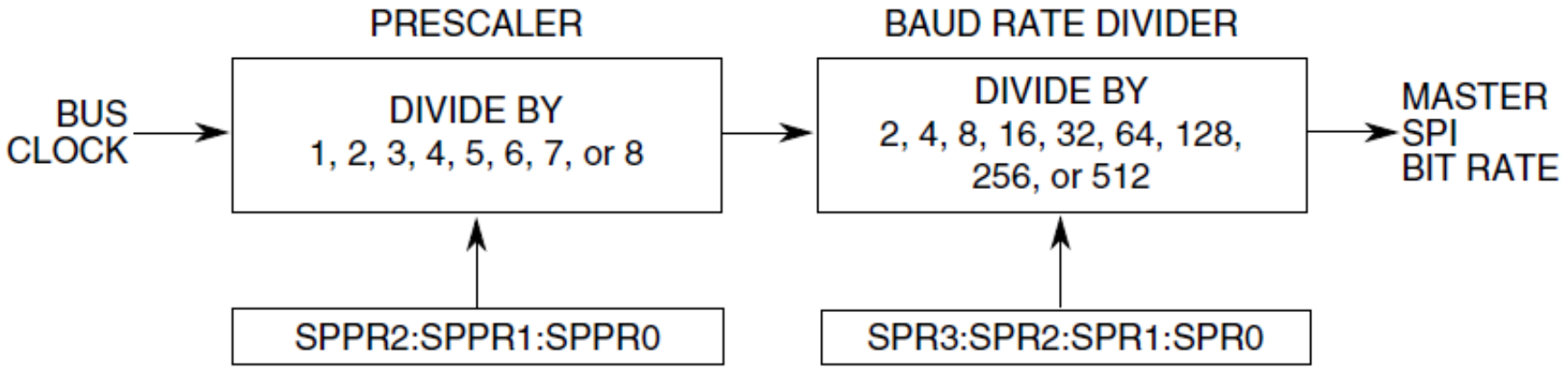
Bit	7	6	5	4	3	2	1	0
Read Write	SPMIE	SPLPIE	TXDMAE	MODFEN	BIDIROE	RXDMAE	SPISWAI	SPC0
Reset	0	0	0	0	0	0	0	0

- SPMIE: SPI interrupt enable for receive data match
- SPLPIE: SPI interrupt enable for wake from low-power mode
- TXDMAE: Transmit DMA enable
- MODFEN: Master mode-fault function enable
- BIDIROE
- RDDMAE: Receive DMA enable
- SPISWAI: Stop SPI in wait mode
- SPC0: Single wire (bidirectional) mode

SPI Baud Rate Register (SPIx_BR)

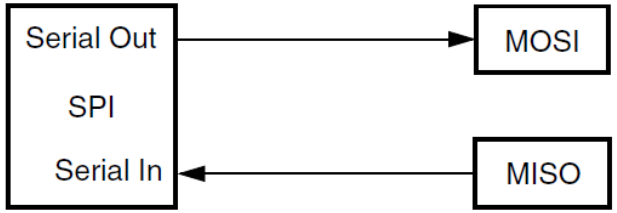
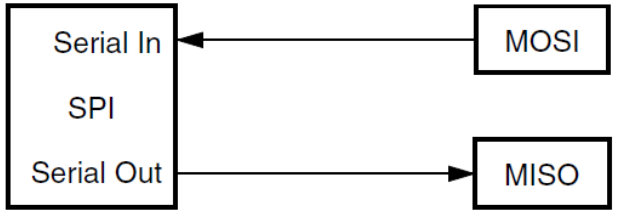
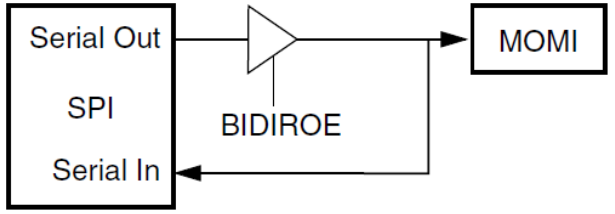
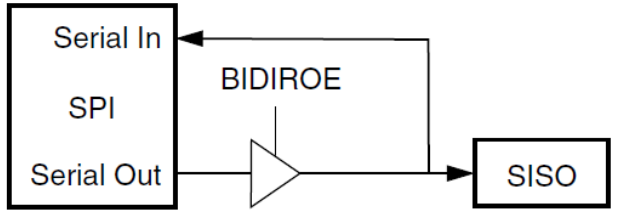
Bit	7	6	5	4	3	2	1	0
Read	0	SPPR[2:0]			SPR[3:0]			
Write								
Reset	0	0	0	0	0	0	0	0

- SPPR: SPI baud rate prescale divisor: divides by n+1
- SPR: SPI baud rate divisor: divides by 2^{n+1}



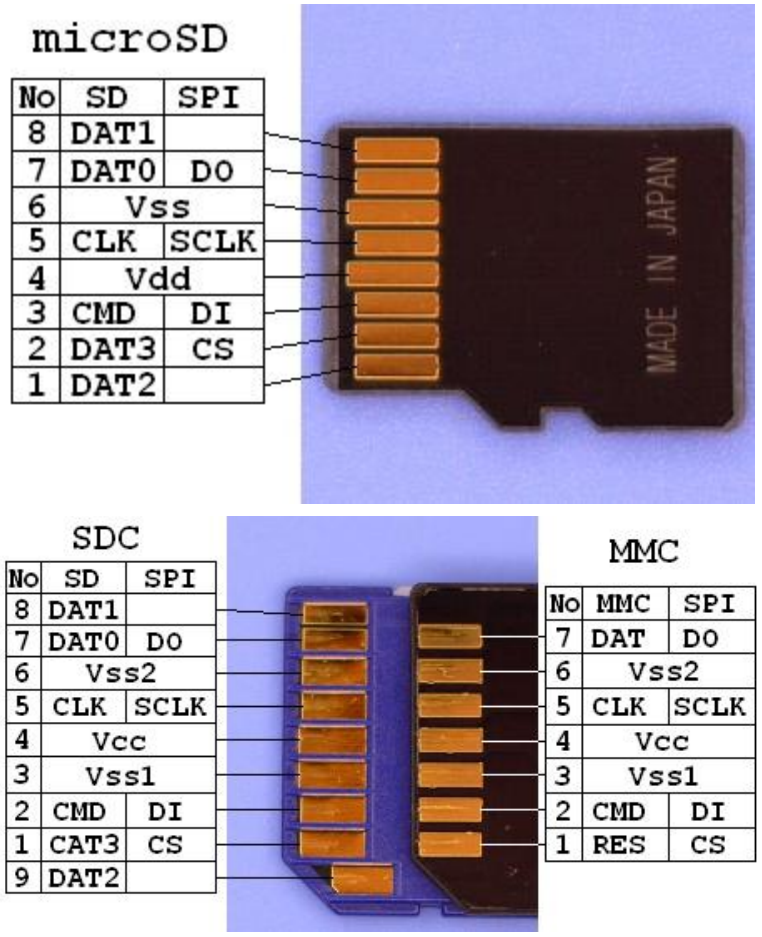
- $f_{SPI} = f_{bus\_clock}/((SPPR+1)*2^{SPR+1})$

Normal and Bidirectional Modes

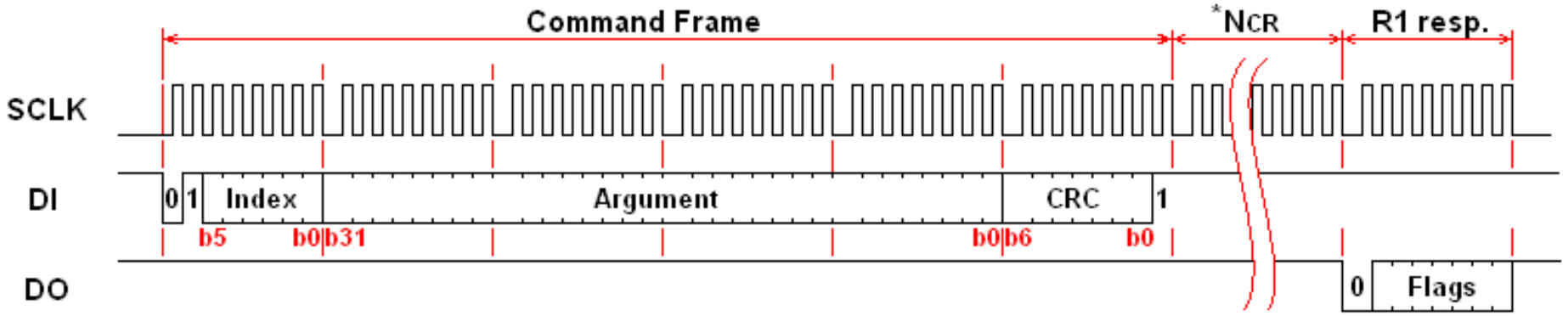
When SPE = 1	Master Mode MSTR = 1	Slave Mode MSTR = 0
Normal Mode **SPC0 = 0**		
Bidirectional Mode **SPC0 = 1**		

SPI Example: Secure Digital Card Access

- SD cards have two communication modes
  - Native 4-bit
  - Legacy SPI 1-bit
- SPI mode 0
  - CPHA=0
  - CPOL=0
- V_{DD} from 2.7 to 3.6 V
- CS: Chip Select (active low)
- Source – FatFS FAT File System Module:
  - http://elm-chan.org/docs/mmc/mmc_e.html
  - http://elm-chan.org/fsw/ff/00index_e.html



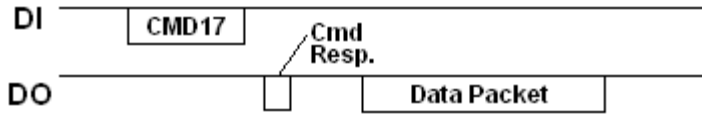
SPI Commands for SD Card



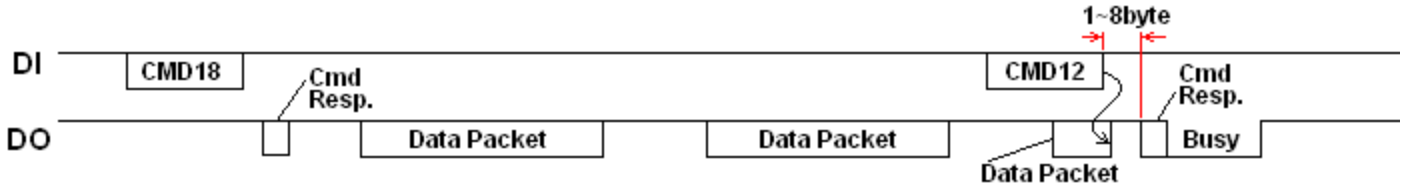
- Host sends a six-byte command packet to card
  - Index, argument, CRC
- Host reads bytes from card until card signals it is ready
  - Card returns
    - 0xff while busy
    - 0x00 when ready without errors
    - 0x01-0x7f when error has occurred

SD Card Transactions

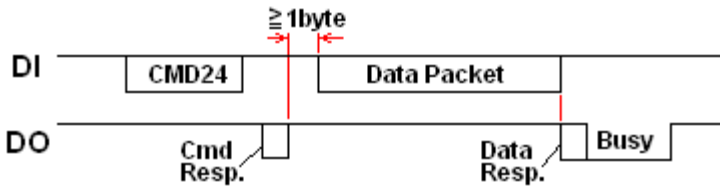
- Single Block Read



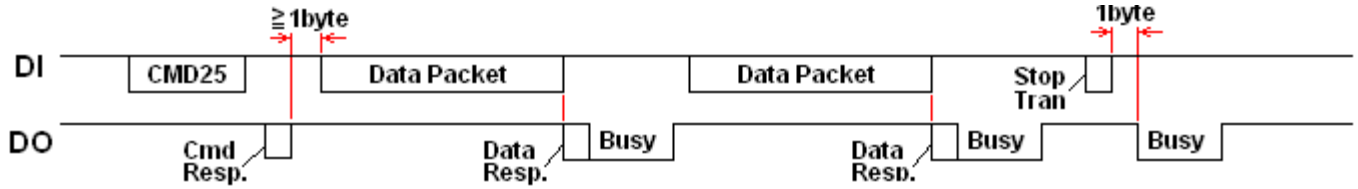
- Multiple Block Rea



- Single Block Write



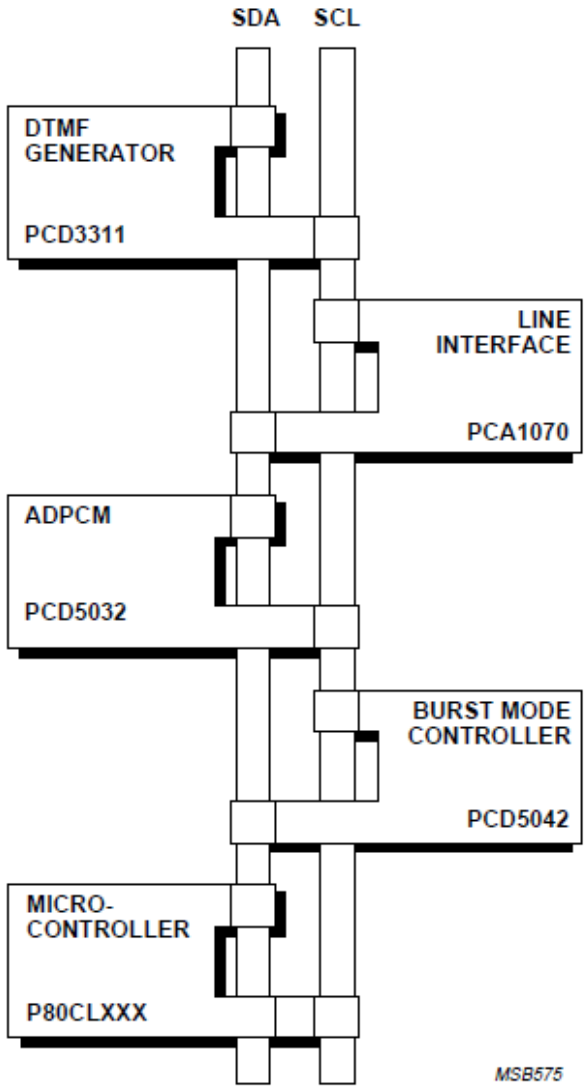
- Multipe Block Write



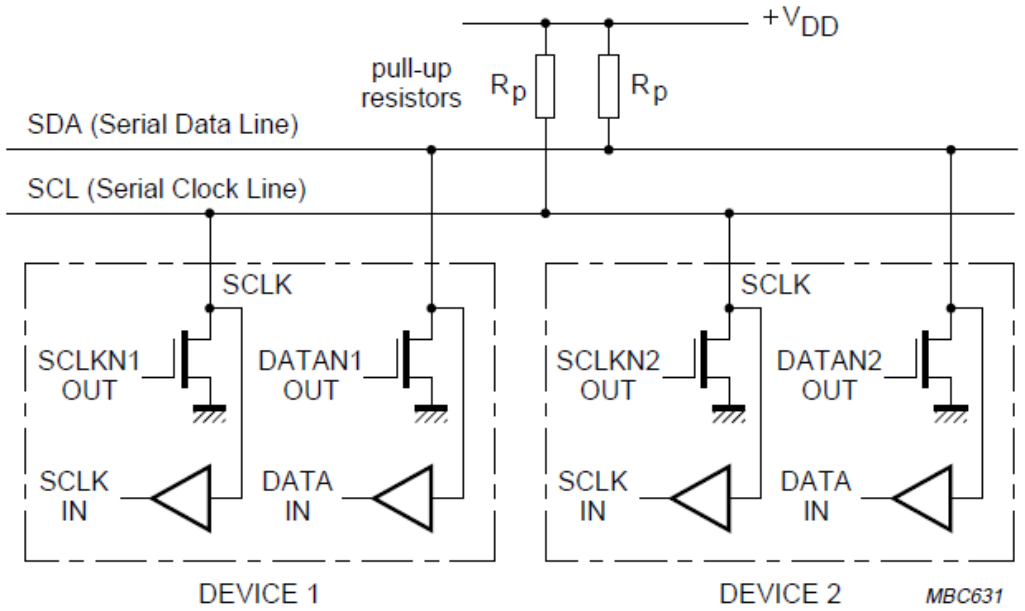
I^2C COMMUNICATIONS

I²C Bus Overview

- "Inter-Integrated Circuit" bus
- Multiple devices connected by a shared serial bus
- Bus is typically controlled by master device, slaves respond when addressed
- I²C bus has two signal lines
  - SCL: Serial clock
  - SDA: Serial data
- Full details available in "The I²C-bus Specification"

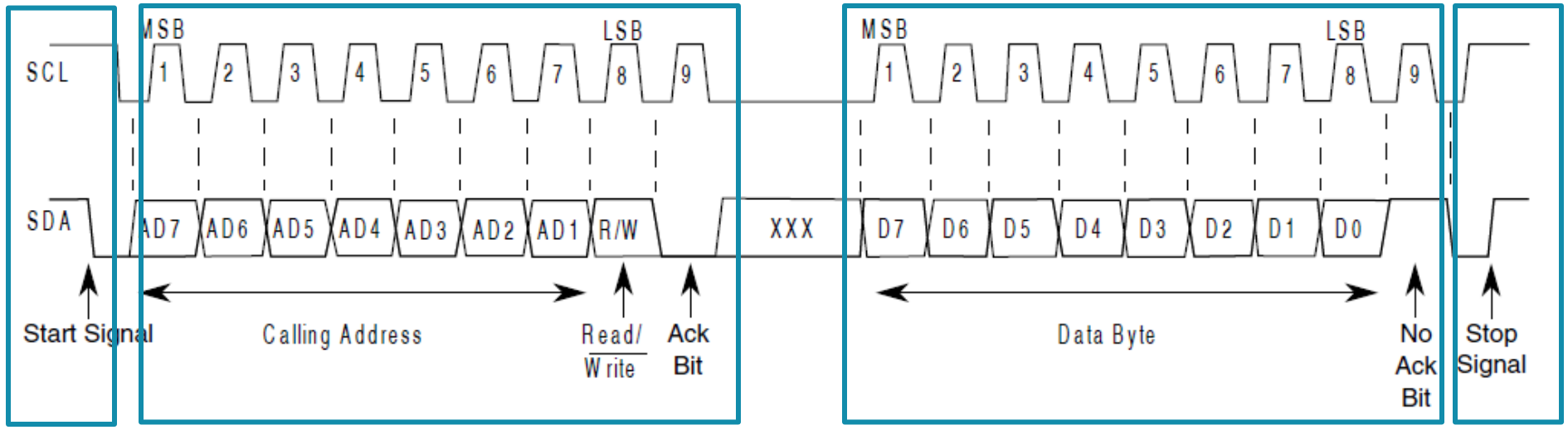


I²C Bus Connections



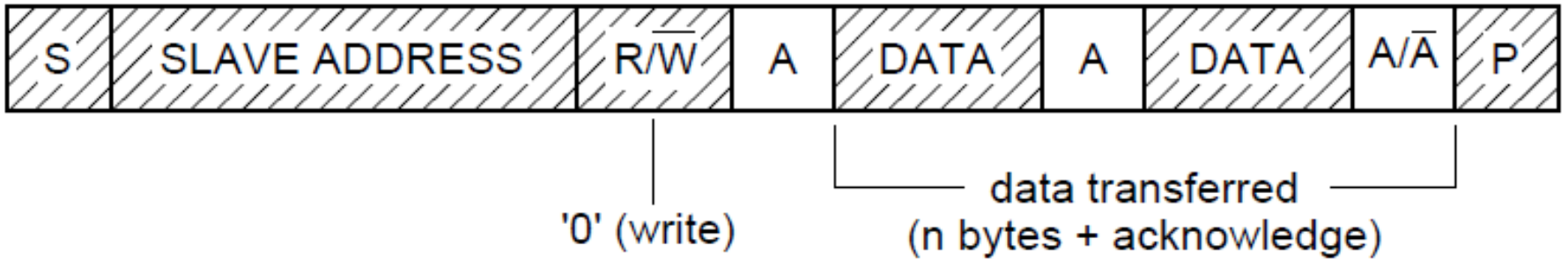
- Resistors pull up lines to V_{DD}
- Open-drain transistors pull lines down to ground
- Master generates SCL clock signal
  - Can range up to 400 kHz, 1 MHz, or more

I²C Message Format



- Message-oriented data transfer with four parts
  1. Start condition
  2. Slave Address transmission
     - Address
     - Command (read or write)
     - Acknowledgement by receiver
  3. Data fields
     - Data byte
     - Acknowledgement by receiver
  4. Stop condition
- Message is made of
  - Signals: Start, Stop, Repeated Start
  - Bytes
  - Acknowledgement bits

Master Writing Data to Slave



from master to slave

from slave to master

MBC605

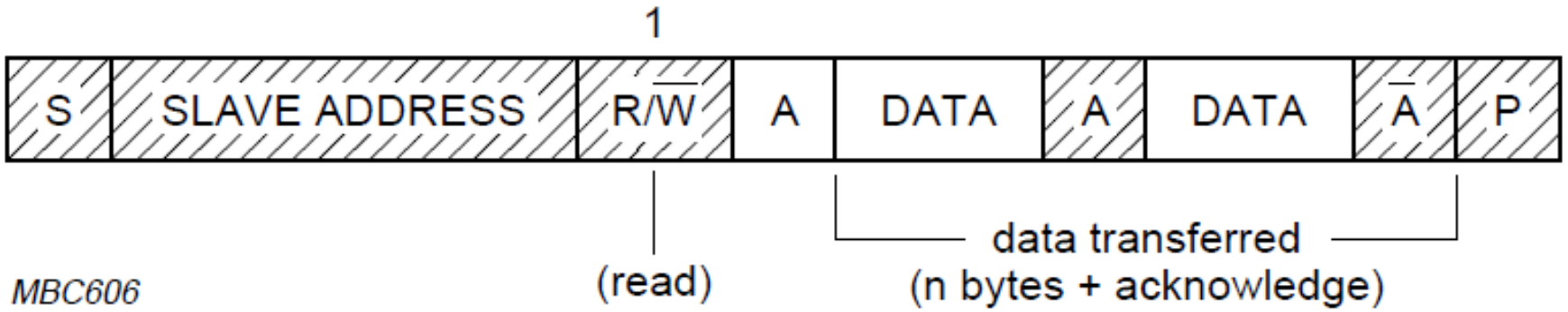
A = acknowledge (SDA LOW)

Ā = not acknowledge (SDA HIGH)

S = START condition

P = STOP condition

Master Reading Data from Slave



MBC606

from master to slave

from slave to master

MBC605

A = acknowledge (SDA LOW)

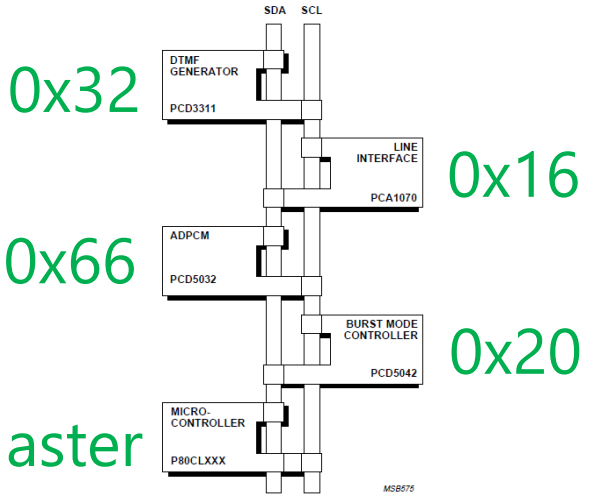
$\overline{A}$ = not acknowledge (SDA HIGH)

S = START condition

P = STOP condition

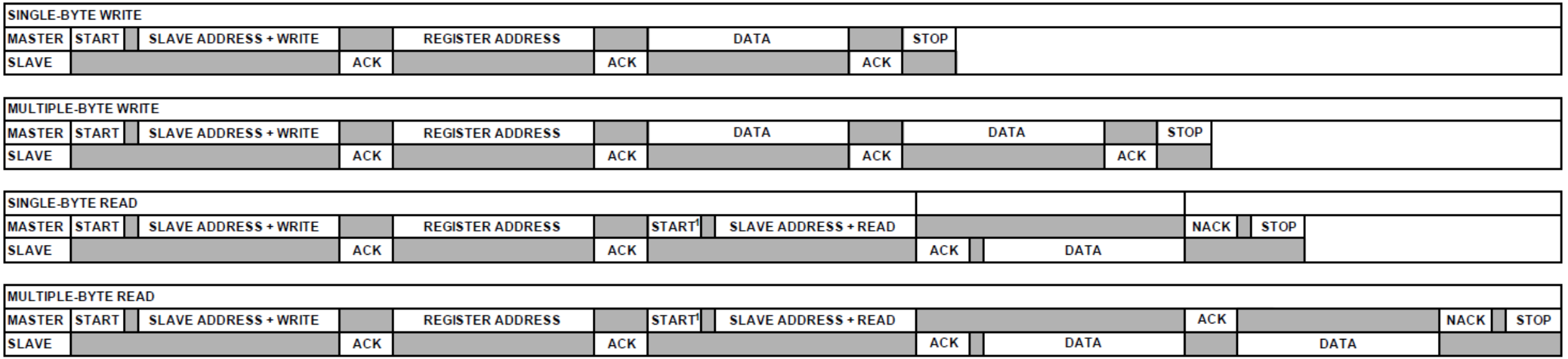
I²C Addressing: Devices and Registers

- Slave device addressing
  - Each slave device has a seven-bit address
  - Can support up to 2^7=128 different devices on same bus
  - Different types of device have different default addresses
  - Sometimes can select a secondary default address by tying a device pin to a different logic level
- Register addressing
  - I2C devices may have multiple control, status, data registers and even data memory internally – how do we get at it?
  - Use the first byte of data as a ***register*** address
  - Example: First seven registers of MMA8451 I2C accelerometer



Name	Type	Register Address	Comment
STATUS/F_STATUS(1)(2)	R	0x00	FMODE = 0, real time status FMODE > 0, FIFO status
OUT_X_MSB(1)(2)	R	0x01	[7:0] are 8 MSBs of 14-bit sample. Root pointer to XYZ FIFO data.
OUT_X_LSB(1)(2)	R	0x02	[7:2] are 6 LSBs of 14-bit real-time sample
OUT_Y_MSB(1)(2)	R	0x03	[7:0] are 8 MSBs of 14-bit real-time sample
OUT_Y_LSB(1)(2)	R	0x04	[7:2] are 6 LSBs of 14-bit real-time sample
OUT_Z_MSB(1)(2)	R	0x05	[7:0] are 8 MSBs of 14-bit real-time sample
OUT_Z_LSB(1)(2)	R	0x06	[7:2] are 6 LSBs of 14-bit real-time sample

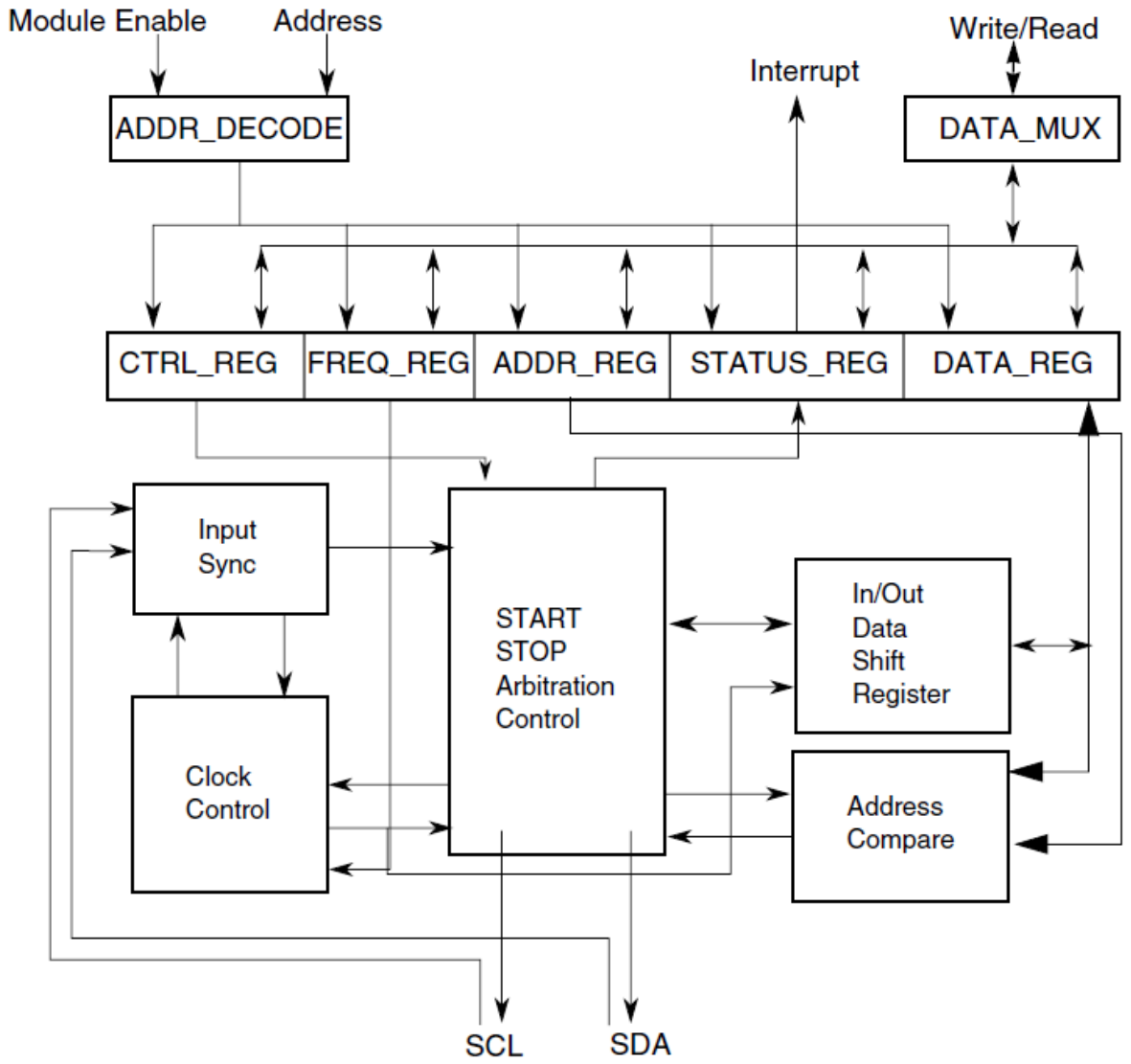
I²C with Register Addressing



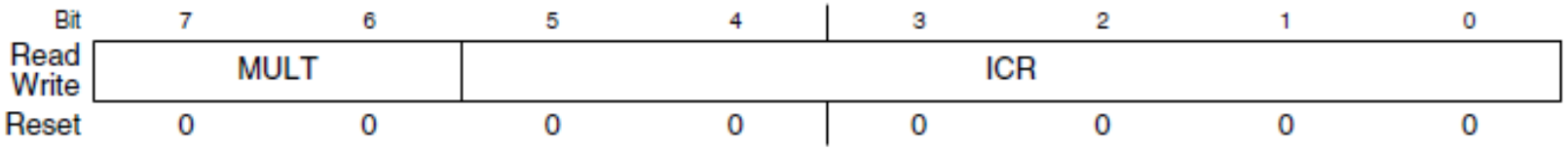
NOTES
1. THIS START IS EITHER A RESTART OR A STOP FOLLOWED BY A START.
2. THE SHADED AREAS REPRESENT WHEN THE DEVICE IS LISTENING.

- Master drives communication
  - Sends start condition, address of slave, read/write command
  - Listens for acknowledgement from slave
  - Sends register address (byte)
  - Listens for acknowledgement from slave

KL25Z I2C Controller



Setting the I²C Baud Rate



Bit	7	6	5	4	3	2	1	0
Read Write	MULT				ICR			
Reset	0	0	0	0	0	0	0	0

- I2Cx_F: Frequency Divider register
  - MULT: specified multiplier mul = 2^{MULT}
    - valid values: 1, 2,4
  - ICR: Clock Rate
  - I2C baud rate = f_{bus}/ (2^{MULT} * ICR)

I2C Control Register 1 – I2Cx_C1

Bit	7	6	5	4	3	2	1	0
Read	IICEN	IICIE	MST	TX	TXAK	0	WUEN	DMAEN
Write						RSTA		
Reset	0	0	0	0	0	0	0	0

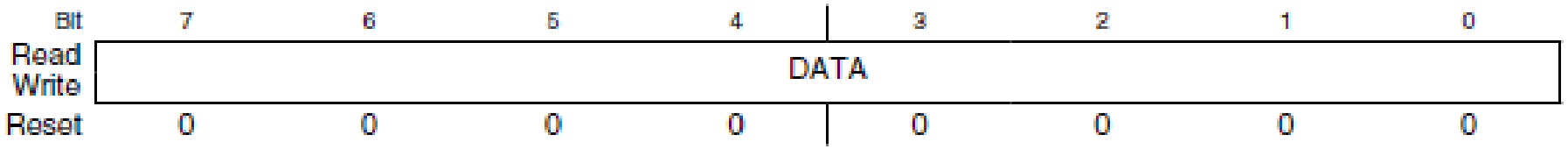
- IICEN - enable I2C module
- IICIE - enable I2C interrupt
- MST - select master mode
  - 0➔1 generates Start condition
  - 1➔0 generates Stop condition
- TX – Select 1 for master transmit and 0 for master receive
- TXAK – Transmit Acknowledge enable
- RSTA – Repeat Start
- WUEN – Wakeup enable
- DMAEN – Enable DMA

I2C Status Register – I2Cx_S

Bit	7	6	5	4	3	2	1	0
Read	TCF	IAAS	BUSY	ARBL	RAM	SRW	IICIF	RXAK
Write				w1c			w1c	
Reset	1	0	0	0	0	0	0	0

- TCF – Transfer Complete flag set after transferring byte and acknowledge bit
- IAAS – Addressed as a Slave
- BUSY – bus busy
- ARBL – arbitration lost
- RAM – Range address match
- SRW – when slave, indicates transmission direction (0: slave receive, 1: slave transmit)
- IICIF: Interrupt pending flag
- RXAK: 0: acknowledge signal received

I2C Data Register – I2Cx_D



Bit	7	6	5	4	3	2	1	0
Read Write	DATA							
Reset	0	0	0	0	0	0	0	0

- 8-bit data register
- Master transmit mode
  - Writing to I2Cx_D starts a data transfer
- Master receive mode
  - Reading from I2Cx_D starts reception of next byte

Macros for Polled Communications

```
#define I2C_M_START  I2C0->C1 |= I2C_C1_MST_MASK
#define I2C_M_STOP   I2C0->C1 &= ~I2C_C1_MST_MASK
#define I2C_M_RSTART I2C0->C1 |= I2C_C1_RSTA_MASK

#define I2C_TRAN     I2C0->C1 |= I2C_C1_TX_MASK
#define I2C_REC      I2C0->C1 &= ~I2C_C1_TX_MASK

#define BUSY_ACK     while(I2C0->S & 0x01)
#define TRANS_COMP   while(!(I2C0->S & 0x80))
#define I2C_WAIT     while((I2C0->S & I2C_S_IICIF_MASK)==0){} \
                                 I2C0->S |= I2C_S_IICIF_MASK;

#define NACK         I2C0->C1 |= I2C_C1_TXAK_MASK
#define ACK          I2C0->C1 &= ~I2C_C1_TXAK_MASK
```

Writing a Single Byte to a Device

```
I2C_TRAN;                   /*set to transmit mode */
I2C_M_START;                /*send start          */
I2C0->D = dev;              /*send dev address */
I2C_WAIT;                   /*wait for ack */

I2C0->D = address;          /*send write address */
I2C_WAIT;

I2C0->D = data;             /*send data */
I2C_WAIT;
I2C_M_STOP;
```

Reading a Single Byte from a Device

```
I2C_TRAN;                   /*set to transmit mode */
I2C_M_START;                /*send start      */
I2C0->D = dev;              /*send dev address      */
I2C_WAIT;                   /*wait for completion */
I2C0->D = address;          /*send read address */
I2C_WAIT;                   /*wait for completion */
I2C_M_RSTART;               /*repeated start */
I2C0->D = (dev|0x1);        /*send dev address (read) */
I2C_WAIT;                   /*wait for completion */
I2C_REC;                    /*set to recieve mode */
NACK;                       /*set NACK after read   */
data = I2C0->D;             /*dummy read      */
I2C_WAIT;                   /*wait for completion */
I2C_M_STOP;                 /*send stop */
data = I2C0->D;             /*read data */
```

Reading Multiple Bytes from a Device: Set Up

```
I2C_TRAN;                   /*set to transmit mode */
I2C_M_START;                /*send start      */
I2C0->D = dev;              /*send dev address */
I2C_WAIT;                   /*wait for ack */
I2C0->D =  address;         /*send read address     */
I2C_WAIT;                   /*wait for completion */
I2C_M_RSTART;               /*repeated start */
I2C0->D = (dev|0x1);        /*send dev address (read) */
I2C_WAIT;                   /*wait for completion */
I2C_REC;                    /*set to receive mode */
```

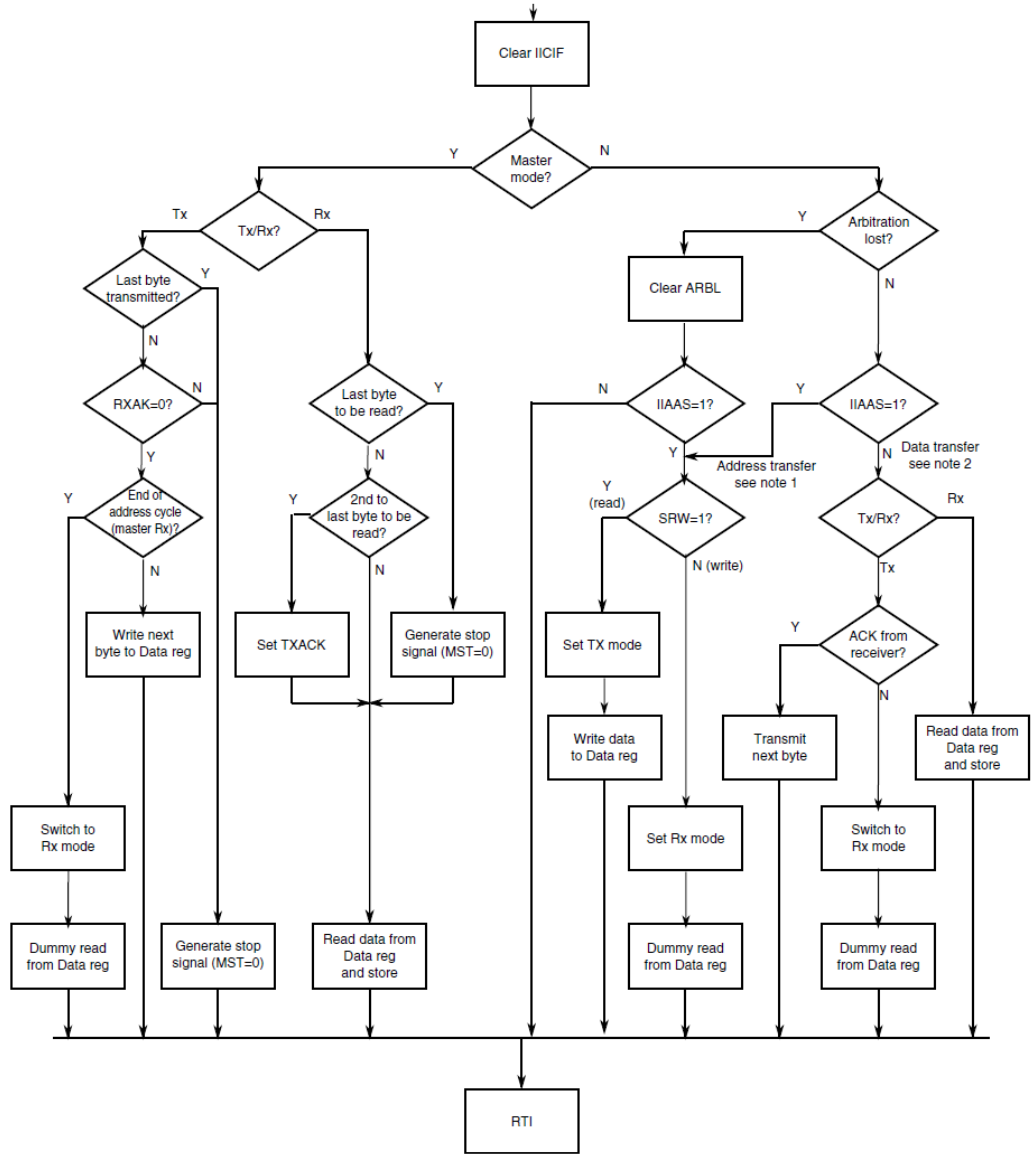
Reading Multiple Bytes from a Device: Data

```
// For each byte
if(isLastRead) {
      NACK;         /*set NACK after read   */
} else {
      ACK;          /*ACK after read  */
}

data = I2C0->D;     /*dummy read      */
I2C_WAIT;           /*wait for completion */

if(isLastRead) {
      I2C_M_STOP; /*send stop */
}
data = I2C0->D;     /*read real data  */
```

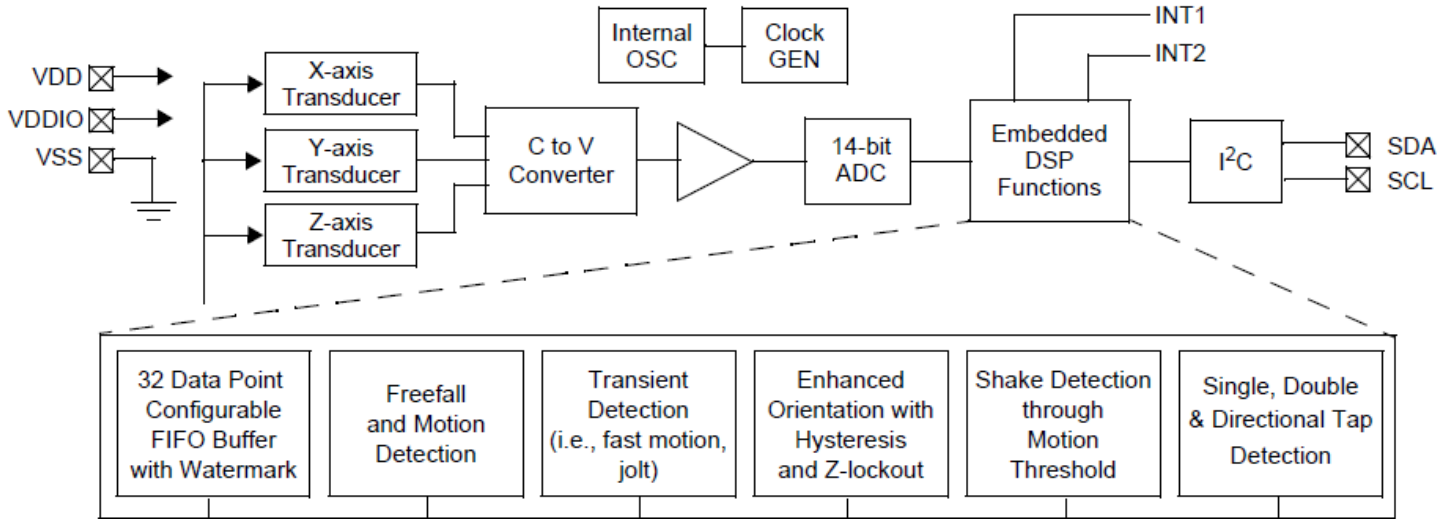
Interrupt-Driven I2C Communications

- Example flowchart from KL25Z Reference Manual chapter on I2C peripheral

- One ISR handles all possible cases
  - Is MCU is in master or slave mode?
  - Arbitration lost (in multi-master bus)?
  - Transmit or receive?
  - More data to send?
  - Acknowledge received?
  - etc.

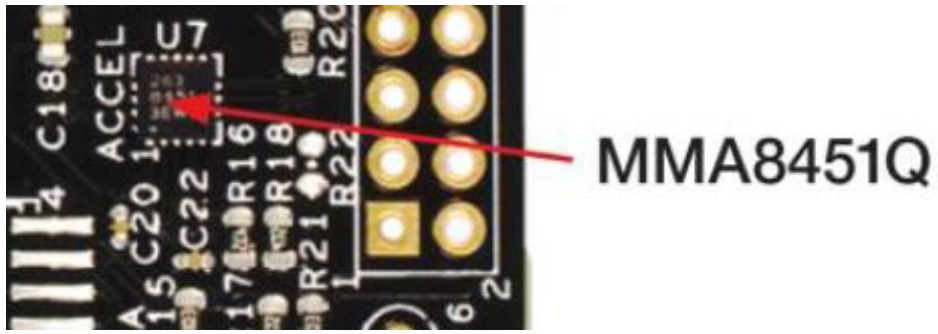


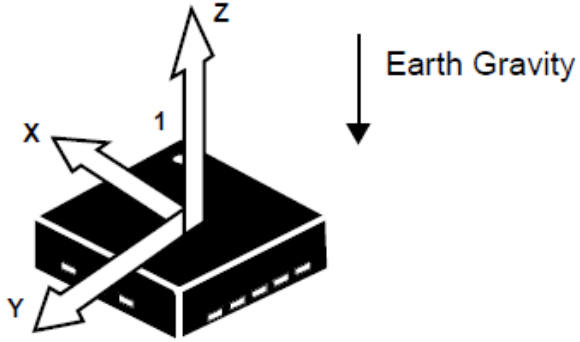
Example I²C Peripheral: 3-Axis Accelerometer

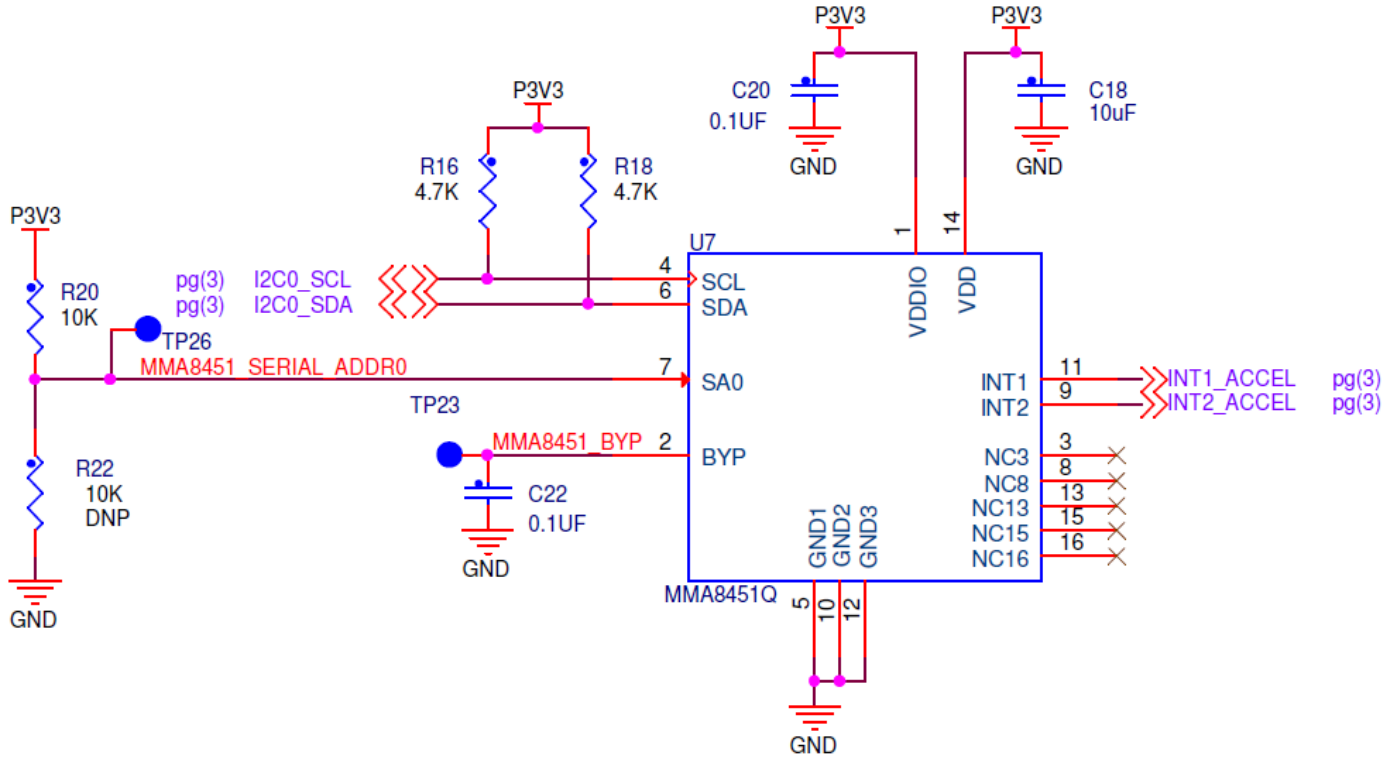


- Freescale MMA8451, included in Freedom KL25Z board
  - Freedom KL26Z board has 3-axis accelerometer and 3-axis magnetometer (compass)
- Can measure acceleration in x/y/z directions up to ±2g, ± 4g, ± 8g
- Can compute rotation about x/y axes (pitch, roll)
- I2C addresses are 0x3A (read) and 0x3B (write)
  - 011101r (r=1 for read, r=0 for write)

MMA8451 on Freedom KL25Z







Main Registers of Interest in MMA8451

- Acceleration Data
  - Signed (two's complement) data
  - 14 or 8 bits per channel (left aligned)
  - X-axis: 0x01, 0x02
  - Y-axis: 0x03, 0x04
  - Z-axis: 0x05, 0x06
- Resolution is about 1/4096 g per LSB (in +/- 2 g mode)
- Who Am I?
  - Used to identify which device this is (IC-specific)
  - Device ID - 0x1A

Name	Type	Register Address	Comment
STATUS/F_STATUS(1)(2)	R	0x00	FMODE = 0, real time status FMODE > 0, FIFO status
OUT_X_MSB(1)(2)	R	0x01	[7:0] are 8 MSBs of 14-bit sample. / Root pointer to XYZ FIFO data.
OUT_X_LSB(1)(2)	R	0x02	[7:2] are 6 LSBs of 14-bit real-time sample
OUT_Y_MSB(1)(2)	R	0x03	[7:0] are 8 MSBs of 14-bit real-time sample
OUT_Y_LSB(1)(2)	R	0x04	[7:2] are 6 LSBs of 14-bit real-time sample
OUT_Z_MSB(1)(2)	R	0x05	[7:0] are 8 MSBs of 14-bit real-time sample
OUT_Z_LSB(1)(2)	R	0x06	[7:2] are 6 LSBs of 14-bit real-time sample

Control Register 1 (0x2A)

0x2A: CTRL_REG1 Register (Read/Write)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ASLP_RATE1	ASLP_RATE0	DR2	DR1	DR0	LNOISE	F_READ	ACTIVE

- ACTIVE: set to 1 to put in active (not standby) mode
- FREAD: set to 1 to fast read just 8 MSBs of XYZ data
- LNOISE: low noise mode
- DR2-0: select output data rate from 1.56 Hz to 800 Hz
- ASLP_RATE1-0: Select Sleep mode rate from 1.56 to 50 Hz

DR 2	DR1	DR0	Output Data Rate
0	0	0	800 Hz
0	0	I	400
0	I	0	200
0	I	I	100
I	0	0	50
I	0	I	12.5
I	I	0	6.25
I	I	I	1.56

Registers for Additional Features

- FIFO configuration
- Interrupt control and status
- Dynamic range
- High-pass filter configuration
- Portrait/Landscape detection settings
- Freefall detection settings
- Sleep control registers
- Offset per axis
- Transient event detection settings
- Tap (pulse) detection settings

Demonstration: Configure the Accelerometer

```
if(i2c_read_byte(MMA_ADDR, REG_WHOAMI) == WHOAMI)      {
      Delay(10);
      //set active, 14 bit data and 100 Hz ODR (0x19)
      i2c_write_byte(MMA_ADDR, REG_CTRL1, 0x01);
      return 1;
}
```

- Source code in mma8451.c
- Basic approach
  - Read byte from the WHOAMI register, verify it matches expected value for MMA8451
  - Delay…
  - Set to active, 14-bit mode, 100 Hz sampling rate

Demonstration: Read the Accelerations

- Send I2C Start condition
- Send read addresses (device and register)
- Read first five bytes of data into data[i]
- Read last byte of data into data[i] (also sends stop condition)
- Append bytes to form 16-bit words (int16_t)
- Divide by four to adjust for scaling

```
i2c_start();
i2c_read_setup(MMA_ADDR , REG_XHI);

for( i=0;i<5;i++)   {
       data[i] =
              i2c_repeated_read(1);
}
data[i] = i2c_repeated_read(0);

for ( i=0; i<3; i++ ) {
       temp[i] = (int16_t)
              ((data[2*i]<<8) |
              data[2*i+1]);
       }


// Right-justify, is 14 bits
acc_X = temp[0]/4;
acc_Y = temp[1]/4;
acc_Z = temp[2]/4;
```

PROTOCOL COMPARISON

Factors to Consider

- How fast can the data get through?
  - Depends on raw bit rate, protocol overhead in packet
- How many hardware signals do we need?
  - May need clock line, chip select lines, etc.
- How do we connect multiple devices (topology)?
  - Dedicated link and hardware per device - point-to-point
  - One bus for master transmit/slave receive, one bus for slave transmit/master receive
  - All transmitters and receivers connected to same bus – multi-point
- How do we address a target device?
  - Discrete hardware signal (chip select line)
  - Address embedded in packet, decoded internally by receiver
- How do these factors change as we add more devices?

Protocol Trade-Offs

Protocol	Speed	Signals Req. for Bidirectional Communication with N devices	Device Addressing	Topology
UART (Point to Point)	Fast – Tens of Mbit/s	2*N (TxD, RxD)	None	Point-to-point full duplex
UART (Multi-drop)	Fast – Tens of Mbit/s	2 (TxD, RxD)	Added by user in software	Multi-drop
SPI	Fast – Tens of Mbit/s	3+N for SCLK, MOSI, MISO, and one SS per device	Hardware chip select signal per device	Multi-point full-duplex, multi-drop half-duplex buses
I^2C	Moderate – 100 kbit/s, 400 kbit/s, 1 Mbit/s, 3.4 Mbit/s. Packet overhead.	2: SCL, SDA	In packet	Multi-point half-duplex bus