Serial Communications
Overview

- Serial communications
  - Concepts
  - Tools
  - Software: polling, interrupts and buffering
- UART communications
  - Concepts
  - STM32F4Discovery UART peripheral
- SPI communications
  - Concepts
  - STM32F4Discovery SPI peripheral
- I²C communications
  - Concepts
  - STM32F4Discovery I²C peripheral
Why Communicate Serially?

- Native word size is multi-bit (8, 16, 32, etc.)

- Often it’s not feasible to support sending all the word’s bits at the same time
  - Cost and weight: more wires needed, larger connectors needed
  - 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 radio transmitters + receivers in the system
Example System: Voyager Spacecraft

- Launched in 1977
- Constraints: Reliability, power, size, weight, reliability, etc.
- “Uplink communications is via S-band (16-bits/sec command rate) while an X-band transmitter provides downlink telemetry at 160 bits/sec normally and 1.4 kbps for playback of high-rate plasma wave data. All data are transmitted from and received at the spacecraft via the 3.7 meter high-gain antenna (HGA).”
  
  [Link](http://voyager.jpl.nasa.gov/spacecraft/index.html)
  
  - Uplink – to spacecraft
  - Downlink – from spacecraft
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
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
- 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 along with the data signal

**Transmitting Device**

**Receiving Device**

Data Sampling Time at Receiver
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*
Synchronous Half-Duplex Serial Data Bus

- Share the serial data line
- Doesn’t allow simultaneous send and receive - is *half-duplex* communication
Asynchronous Serial Communication

- Eliminate the clock line!
- Transmitter and receiver must generate clock *locally*
- Transmitter must add start bit (always same value) to indicate start of each data frame
- 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 \( T_{\text{bit}} \times (N+1.5) \)
- Stop bit is also used to detect some timing errors
Serial Communication Specifics

- **Data frame fields**
  - Start bit (one bit)
  - Data (LSB first or MSB, 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!

- Saelae 8-Channel Logic Analyzer
  - $150 (www.saelae.com)
  - Plugs into PC's USB port
  - Decodes SPI, asynchronous serial, I²C, 1-Wire, CAN, etc.
  - Build your own: with Logic Sniffer or related open-source project
SOFTWARE STRUCTURE – 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)
  - Transmit ISR – executes when serial interface is ready to send another character
  - Receive ISR – executes when serial interface receives a character
  - Error ISR(s) – 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-M4 provides various IRQs for all communication interface's events
  - I2C1EVENT
  - I2C1ERROR
  - I2C2EVENT
  - I2C2ERROR
  - SPI1
  - SPI2
  - USART1
  - USART2
  - USART3

- Decide which operation to take in each IRQ

```c
void USART2_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
  - tx ISR unloads tx_q
  - rx ISR loads rx_q
- 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
#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

```c
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

```c
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 STRUCTURE – PARSING MESSAGES
Decoding Messages

- **Two types of messages**
  - **Actual binary data** sent
    - First identify message type
    - Second, 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
    - First identify message type
    - Second, based on this message type, translate (parse) the data from the ASCII message format into a binary format
    - Third, copy the binary data into variables
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

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

### Table A.53 Output ID and Packet Description

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


$GPRMC,hmmss.ss,A,1111.11,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

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

hh Checksum (Mandatory for RMC)
State Machine for Parsing NMEA-0183

- **Start**
  - Append char to buf.
  - $**$ Any char. except *, \r or \n, non-text, or counter>6

- **Talker + Sentence Type**
  - Append char to buf.
  - Inc. counter
  - buf==$SDBBT, $VWVHW, or $YXXDR
  - Enqueue all chars. from buf

- **Sentence Body**
  - Any char. except *
  - Enqueue char

- **Checksum 1**
  - Any char.
  - Save as checksum1

- **Checksum 2**
  - Any char.
  - Save as checksum2
Parsing

```c
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;
```
STM32F407VG AND DISCOVERY SPECIFICS
Discovery Serial I/O

- For full detail, refer to DM00039084 the User Manual of Discovery board.
Clock Gating for Serial Comm.

- SPI 1,2; USART 2,3; UART 4,5 and I2C 1,2,3 are clocked through APB1
- USART 1 and 6 are clocked through APB2
- Enable the clock before using these serial communication modules.
UNIVERSAL SYNCHRONOUS ASYNCHRONOUS RECEIVER TRANSMITTER (USART)
USART Block Diagram

USARTDIV = DIV Mantissa + (DIV Fraction / 8 × (2 - OVER8))
Like other modules, USART in the STM32F4 families is capable to operate in many modes.

<table>
<thead>
<tr>
<th>USART modes</th>
<th>USART1</th>
<th>USART2</th>
<th>USART3</th>
<th>USART4</th>
<th>USART5</th>
<th>USART6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous mode</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Hardware flow control</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>NA</td>
<td>NA</td>
<td>X</td>
</tr>
<tr>
<td>Multibuffer communication (DMA)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Multiprocessor communication</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Synchronous</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>NA</td>
<td>NA</td>
<td>X</td>
</tr>
<tr>
<td>Smartcard</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>NA</td>
<td>NA</td>
<td>X</td>
</tr>
<tr>
<td>Half-duplex (single-wire mode)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IrDA</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>LIN</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

This slide will introduce the Asynchronous mode, which is sometimes referred as Universal asynchronous receiver/transmitter (UART).
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, since the only timing reference is the start bit
How the STM32F4 works

- Transmitter and receiver must agree on several things (protocol)
  - Order of data bits
    - LSB will be transmitted first
  - Number of data bits
    - Can be configured as 8 or 9 bits
  - What a start bit is (1 or 0)
    - 0
  - What a stop bit is (1 or 0)
    - 1
    - Configurable length (0.5, 1, 1.5 or 2)
  - How long a bit lasts
    - Software programmable phase and polarity

- Many of them are configurable as well
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

- FTM32F4 provides configurable oversampling rate of either 8 or 16 times the baud rate clock

- Two voting method: Single sample in the center or majority vote of the three samples in the center
Fractional Baud Rate Generation

- Both Rx and Tx are set to the same baud rate as programmed in the Mantissa and Fraction values of USARTDIV.

In standard USART (SPI mode included):
- Baud rate = \( \frac{f_{ck}}{\text{UART Module Clock} \times 16} \) 
- Example: derive USARTDIV value from USART_BRR (OVER8=0)
  - Mantissa = 0d27, Fraction = 12/16 = 0d0.75 (left shift by 4 bits)
  - Therefore the USARTDIV = 0d27.75

Example: derive USARTDIV value from USART_BRR (OVER8=0)
- Mantissa = 0d27, Fraction = 12/16 = 0d0.75 (left shift by 4 bits)
  - Therefore the USARTDIV = 0d27.75
Using the UART

Transmitter

- Configure the GPIO
  - AF mode, fast speed
- Enable the USART (Set UE in CR1)
- Define word length (writing M bit in CR1)
- Program the stop bits (CR2)
- Using DMA? (DMAT in CR3)
- Parity Check? (PCE/PS in CR1)
- Configure Baud Rate (USART_BRR)
- First transmission (Set TE in CR1)
- Write data to send (DR)
- Repeat writing data to send (DR)
- For ending the transmission, wait until the last frame is complete (when TC=1)

Receiver

- Configure the GPIO
  - AF mode, fast speed
- Enable the USART (Set UE in CR1)
- Define word length (M bit in CR1)
- Program the stop bits (CR2)
- Using DMA? (DMAR in CR3)
- Parity Check? (PCE/PS in CR1)
- Configure Baud Rate (USART_BRR)
- Begin waiting for start bit (Set RE in CR1)
- If RXNE is set, then the data has been received and can be read
- Interrupt if RXNEIE bit is set
USART Control Register 1 (USART_CR1)

- OVER8: 0 oversampling by 16; 1 oversampling by 8
- UE: Write 1 to enable
- M: Word length: 0 8 data bits; 1 9 data bits
- M: Select 9-bit data mode (instead of 8-bit data)
- WAKE: Wakeup method
- PCE: Parity enabled with 1
- PS: Odd parity with 1, even parity with 0
- TE: Transmitter enable
- RE: Receiver enable
**USART Control Register 2 and 3 (USART_CR2/3)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved</td>
<td>-</td>
</tr>
<tr>
<td>30</td>
<td>Reserved</td>
<td>-</td>
</tr>
<tr>
<td>29</td>
<td>Reserved</td>
<td>-</td>
</tr>
<tr>
<td>28</td>
<td>Reserved</td>
<td>-</td>
</tr>
<tr>
<td>27</td>
<td>Reserved</td>
<td>-</td>
</tr>
<tr>
<td>26</td>
<td>Reserved</td>
<td>-</td>
</tr>
<tr>
<td>25</td>
<td>Reserved</td>
<td>-</td>
</tr>
<tr>
<td>24</td>
<td>Reserved</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>Reserved</td>
<td>-</td>
</tr>
<tr>
<td>22</td>
<td>Reserved</td>
<td>-</td>
</tr>
<tr>
<td>21</td>
<td>Reserved</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>Reserved</td>
<td>-</td>
</tr>
<tr>
<td>19</td>
<td>Reserved</td>
<td>-</td>
</tr>
<tr>
<td>18</td>
<td>Reserved</td>
<td>-</td>
</tr>
<tr>
<td>17</td>
<td>Reserved</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>Reserved</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>LINEN</td>
<td>rw</td>
</tr>
<tr>
<td>14</td>
<td>STOP[1:0]</td>
<td>rw</td>
</tr>
<tr>
<td>13</td>
<td>CLKEN</td>
<td>rw</td>
</tr>
<tr>
<td>12</td>
<td>CPOL</td>
<td>rw</td>
</tr>
<tr>
<td>11</td>
<td>CPHA</td>
<td>rw</td>
</tr>
<tr>
<td>10</td>
<td>LBCL</td>
<td>rw</td>
</tr>
<tr>
<td>9</td>
<td>Reserved</td>
<td>rw</td>
</tr>
<tr>
<td>8</td>
<td>LBDIE</td>
<td>rw</td>
</tr>
<tr>
<td>7</td>
<td>LBDL</td>
<td>rw</td>
</tr>
<tr>
<td>6</td>
<td>Reserved</td>
<td>rw</td>
</tr>
<tr>
<td>5</td>
<td>DMAR</td>
<td>rw</td>
</tr>
<tr>
<td>4</td>
<td>SCEN</td>
<td>rw</td>
</tr>
<tr>
<td>3</td>
<td>NACK</td>
<td>rw</td>
</tr>
<tr>
<td>2</td>
<td>HDSEL</td>
<td>rw</td>
</tr>
<tr>
<td>1</td>
<td>IRLP</td>
<td>rw</td>
</tr>
<tr>
<td>0</td>
<td>IREN</td>
<td>rw</td>
</tr>
<tr>
<td></td>
<td>EIE</td>
<td>rw</td>
</tr>
</tbody>
</table>

**STOP: STOP bits:**
- 00 1 Stop bit; 01 0.5 Stop bit; 10 2 Stop bits 11 1.5 Stop bit

**ONEBIT: Sample method**
- 0: Three sample bit method; 1: One sample bit method

**DMAT: DMA enable transmitter**

**DMAR: DMA enable Reciever**
### Baud Rate Register (USART_BRR)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DIV_Mantissa[11:0]</strong></td>
<td>Mantissa for the USART Divider</td>
</tr>
<tr>
<td><strong>DIV_Fraction[3:0]</strong></td>
<td>Fraction of USART Divider</td>
</tr>
<tr>
<td></td>
<td>When OVER8 is set, DIV_Fraction3 bit is not considered and must be cleared</td>
</tr>
</tbody>
</table>

- **DIV_Mantissa[11:0]**
  - Mantissa for the USART Divider
- **DIV_Fraction[3:0]**
  - Fraction of USART Divider
  - When OVER8 is set, DIV_Fraction3 bit is not considered and must be cleared
Data Register

- **DR[8:0]: Data value**
  - The data to be transmitted or the data received, depending on whether it is read from or written to.

- The TDR register provides the parallel interface between the internal bus and the output shift register.
- The RDR register provides the parallel interface between the input shift register and the internal bus.
UART Status Register (USART_SR)

- TXE: Transmit data register empty
- TC: Transmission complete.
- IDLE: IDLE line detected
- ORE: 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.
- PE: Parity error. Incorrect parity received.
Software for Polled Serial Comm.

```c
void usart2_init(uint32_t pclk, uint32_t baudrate){
    uint32_t temp=0x00;
    uint32_t integer=0x00;
    uint32_t fraction=0x00;
    integer=((25*pclk*1000000)/(4*baudrate));
    temp=((integer/100)<<4);
    fraction=integer-(100*(temp>>4));
    temp=(((fraction*16)+50)/100)&((uint8_t) 0x0F);

    RCC->AHB1ENR|=RCC_AHB1ENR_GPIOAEN;
    RCC->APB1ENR|=RCC_APB1ENR_USART2EN;
    GPIOA->MODER|=GPIO_MODER_MODER2_1;//Pin2 mode AF
    //GPIOA->PUPDR|=GPIO_PUPDR_PUPDR2_0;//Pull up
    GPIOA->OSPEEDR|=GPIO_OSPEEDER_OSPEEDR2_1;

    GPIOA->AFR[0]|=0x00000700;//Set the AF to AF7(USART1~3);

    SET_BIT(RCC->AHB1RSTR,RCC_APB1RSTR_USART2RST);
    CLEAR_BIT(RCC->AHB1RSTR,RCC_APB1RSTR_USART2RST);

    USART2->CR1|=USART_CR1_UE;
    USART2->BRR=temp;
    USART2->CR1|=USART_CR1_TE;}
```
Software for Polled Serial Comm.

```c
void usart2_send(uint16_t data){
    USART2->DR=(data&(uint16_t)0x01ff);
    while(!READ_BIT(USART2->SR, USART_SR_TC)){}
}

uint16_t usart3_receive(){
    while(!READ_BIT(USART3->SR, USART_SR_RXNE)){}
    return (uint16_t)(USART3->DR & (uint16_t)0x01FF);
}
```
Example Transmitter

```c
while(1){
    usart2_send(data);
}
```
Example Receiver

while (1) {
    echo=usart3_receive();
}

- Also possible to display the received data on LCD screen.
  - Using the module we built in the previous lectures
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
### USART Control Register 1 (USART_CR1)

<table>
<thead>
<tr>
<th>Bit 31-16</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-29</td>
<td>RESERVED</td>
</tr>
<tr>
<td>28</td>
<td>RES</td>
</tr>
<tr>
<td>27</td>
<td>WAKE</td>
</tr>
<tr>
<td>26-25</td>
<td>PCE</td>
</tr>
<tr>
<td>24-23</td>
<td>PS</td>
</tr>
<tr>
<td>22-21</td>
<td>PEIE</td>
</tr>
<tr>
<td>20-19</td>
<td>TXEIE</td>
</tr>
<tr>
<td>18-17</td>
<td>TCIE</td>
</tr>
<tr>
<td>16</td>
<td>RXNEIE</td>
</tr>
<tr>
<td>15-14</td>
<td>IDLEIE</td>
</tr>
<tr>
<td>13-12</td>
<td>TE</td>
</tr>
<tr>
<td>11-10</td>
<td>RE</td>
</tr>
<tr>
<td>9-8</td>
<td>RWU</td>
</tr>
<tr>
<td>7-6</td>
<td>SBK</td>
</tr>
</tbody>
</table>

- **TCIE**: Transmission complete interrupt enable
- **RXNEIE**: RXNE interrupt enable, this means whenever overrun error or read data register is not empty (ORE=1 or RXNE=1), it will assert the interrupt.
Interrupt Initialization (transmitter)

```c
void Init_UART2(uint32_t pclk, uint32_t baud_rate) {

    NVIC_SetPriority(USART2_IRQn, 0);
    NVIC_ClearPendingIRQ(USART2_IRQn);
    NVIC_EnableIRQ(USART2_IRQn);

    USART2->CR1|=USART_CR1_TCIE;

    ...
}
```
Interrupt Handler: Transmitter

```c
void UART2_IRQHandler(void) {
    NVIC_ClearPendingIRQ(USART2_IRQn);
    usart2_send(echo-1);
    ...
}
```
Interrupt Handler: Receiver

```c
void UART2_IRQHandler(void) {
    ...
    if (READ_BIT(USART2->SR, USART_SR_RXNE)) {
        // received a character
    }
}
```
Interrupt Handler: Error Cases

```c
void UART2_IRQHandler(void) {
    ...
    if (READ_BIT(USART2->SR, USART_SR_ORE)) {
        // handle the error
        // clear the flag
    }
}
```
Example UART Application

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

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

### Table 3.1 Default Protocols and Port Configurations

<table>
<thead>
<tr>
<th>Port</th>
<th>Input Protocol</th>
<th>Default Setup</th>
<th>Output Language</th>
<th>Default Setup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TSIP</td>
<td>Baud Rate: 9600 Data Bits: 8 Parity: Odd Stop Bits: 1 No Flow Control</td>
<td>TSIP</td>
<td>Baud Rate: 9600 Data Bits: 8 Parity: Odd Stop Bits: 1 No Flow Control</td>
</tr>
<tr>
<td>2</td>
<td>RTCM</td>
<td>Baud Rate: 4800 Data Bits: 8 Parity: None Stop Bits: 1 No Flow Control</td>
<td>NMEA</td>
<td>Baud Rate: 4800 Data Bits: 8 Parity: None Stop Bits: 1 No Flow Control</td>
</tr>
</tbody>
</table>
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
  - Can also supply 5 V, 3.3 V from USB
Building on Asynchronous Comm.

- **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 3.3V 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
Demonstration: Echo back

- Transmit bytes through USART2 and received by USART3
- After successful receiving a byte, decrement and send it to DR of USART2 to send again
- No interrupt, pulling only
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

- **SPI interface of STM32F4 also supports I²S audio protocol.**
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

---

**SS** is also referred as NSS in some documents.
Using the SPI

- **Slave mode**
  - Decide data frame format (DFF)
  - Select the relationship (CPOL/CPHA)
  - MSB or LSB first? (LSBFIRST in CR1)
  - Using DMA? (DMAT in CR3)
  - Handle the NSS or SSM and SSI bit depending on the mode
  - TI mode protocol? (FRF in CR2)
  - Clear the MSTR and set SPE in CR1

- **Master mode**
  - Baud rate (BR in CR1)
  - Select the relationship (CPOL/CPHA)
  - Decide data frame format (DFF)
  - MSB or LSB first? (LSBFIRST in CR1)
  - Handle the NSS or SSM and SSI bit depending on the mode
  - TI mode protocol? (FRF in CR2)
  - Set MSTR and SPE in CR1

- **MOSI is input and MISO is output**

  **Transmit**
  - Parallel-load data byte into Tx buffer during a write cycle
  - Transfer data from buffer to shift register

  **Receive**
  - Transfer data from shift register to Rx buffer and set the RXNE flag

- **MOSI is output and MISO is input**

  **Transmit**
  - Write a byte into Tx buffer
  - Transfer data from buffer to shift register

  **Receive**
  - Transfer data from shift register to RX buffer and set the RXNE flag
## SPI Control Register 1 (SPI_CR1)

<table>
<thead>
<tr>
<th>Bit 15</th>
<th>Bit 14</th>
<th>Bit 13</th>
<th>Bit 12</th>
<th>Bit 11</th>
<th>Bit 10</th>
<th>Bit 9</th>
<th>Bit 8</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIDI MODE</td>
<td>BIDI OE</td>
<td>CRC EN</td>
<td>CRC NEXT</td>
<td>DFF</td>
<td>RX ONLY</td>
<td>SSM</td>
<td>SSI</td>
<td>LSB FIRST</td>
<td>SPE</td>
<td>BR [2:0]</td>
<td>MSTR</td>
<td>CPOL</td>
<td>CPHA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
</tr>
</tbody>
</table>

- **DFF**: data frame format
  - 0: 8-bit; 1: 16-bit
- **LSBFIRST**: frame format
  - 0: MSB first; 1: LSB first
- **SPI**: SPI enable
- **BR[2:0]**: Baud rate control
  - 000: \( f_{PCLK}/2 \)
  - 001: \( f_{PCLK}/4 \)
  - 010: \( f_{PCLK}/8 \)
  - 011: \( f_{PCLK}/16 \)
  - 100: \( f_{PCLK}/32 \)
  - 101: \( f_{PCLK}/64 \)
  - 110: \( f_{PCLK}/128 \)
  - 111: \( f_{PCLK}/256 \)
- **MSTR**: Master selection
SPI Control Register 2 (SPI_CR2)

- **TXEIE**: Tx buffer empty interrupt enable
- **RXNEIE**: Rx buffer not empty interrupt enable
- **ERRIE**: Error interrupt enable
- **FRF**: Frame format
  - 0: SPI Motorola mode
  - 1: SPI TI mode
- **SSOE**: SS output enable
- **TXDMAEN**: Tx buffer DMA enable
- **RXDMAEN**: Rx buffer DMA enable
Clock and Phase Settings: CPHA = 1
Clock and Phase Settings: CPHA = 0
### SPI Status Register (SPI_SR)

<table>
<thead>
<tr>
<th></th>
<th>SPI Status Register (SPI_SR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Reserved</td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>FRE</td>
</tr>
<tr>
<td>7</td>
<td>BSY</td>
</tr>
<tr>
<td>6</td>
<td>OVR</td>
</tr>
<tr>
<td>5</td>
<td>MODF</td>
</tr>
<tr>
<td>4</td>
<td>CRC ERR</td>
</tr>
<tr>
<td>3</td>
<td>UDR</td>
</tr>
<tr>
<td>2</td>
<td>CHSID E</td>
</tr>
<tr>
<td>1</td>
<td>TXE</td>
</tr>
<tr>
<td>0</td>
<td>RXNE</td>
</tr>
</tbody>
</table>

- **FRE**: Frame format error
- **BSY**: Busy flag
- **OVR**: Overrun flag
- **MODF**: Mode fault
- **UDR**: Underrun flag
- **TXE**: Transmit buffer empty
- **RXNE**: Receive buffer not empty
# Normal and Bidirectional Modes

<table>
<thead>
<tr>
<th>When SPE = 1</th>
<th>Master Mode MSTR = 1</th>
<th>Slave Mode MSTR = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Normal Mode</strong>&lt;br&gt;SPC0 = 0</td>
<td><img src="normal_mode_diagram" alt="Diagram" /></td>
<td><img src="slave_mode_diagram" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="normal_mode_diagram" alt="Normal Mode Diagram" /></td>
<td><img src="normal_mode_diagram" alt="Normal Mode Diagram" /></td>
<td><img src="normal_mode_diagram" alt="Normal Mode Diagram" /></td>
</tr>
<tr>
<td><strong>Bidirectional Mode</strong>&lt;br&gt;SPC0 = 1</td>
<td><img src="bidirectional_mode_diagram" alt="Diagram" /></td>
<td><img src="bidirectional_mode_diagram" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="bidirectional_mode_diagram" alt="Bidirectional Mode Diagram" /></td>
<td><img src="bidirectional_mode_diagram" alt="Bidirectional Mode Diagram" /></td>
<td><img src="bidirectional_mode_diagram" alt="Bidirectional Mode Diagram" /></td>
</tr>
</tbody>
</table>
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/docs/mmc/mmc_e.html)
  - [http://elm-chan.org/fsw/ff/00index_e.html](http://elm-chan.org/fsw/ff/00index_e.html)
SPI Commands

- 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 Read**

- **Single Block Write**

- **Multiple Block Write**

---

**SD Card Transactions**

- **Single Block Read**

  - DI: CMD17 Cmd Resp.
  - DO: Data Packet

- **Multiple Block Read**

  - DI: CMD18 Cmd Resp.
  - DO: Data Packet

- **Single Block Write**

  - DI: CMD24 Data Packet
  - DO: Data Resp. Busy

- **Multiple Block Write**

  - DI: CMD25 Data Packet Data Packet Data Packet Stop Tran
  - DO: Data Resp. Busy Data Resp. Busy Busy
I²C 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
**Master Writing Data to Slave**

- **S**: Slave Address
- **R/W**: Read/Write
- **A**: Acknowledge
- **DATA**: Data
- **A/Ā**: Not Acknowledge
- **P**: Stop condition

Data transferred (n bytes + acknowledge)

- '0' (write)
- From master to slave

- A = acknowledge (SDA LOW)
- Ā = not acknowledge (SDA HIGH)
- S = START condition
- P = STOP condition

*MBC605*
Master Reading Data from Slave

1

S | SLAVE ADDRESS | R/W | A | DATA | A | DATA | A | P

(read)

data transferred (n bytes + acknowledge)

MBC606
I²C with Register Addressing

### 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

**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.
I^2C Addressing

- Each device (IC) has seven-bit address
  - 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

- What if we treat the first byte of data as a register address?
  - Can specify registers *within* a given device: eight bits
  - Example: Accelerometer has up to 58 registers
STM32F40x/41x I²C Controller

Data register

Comparator

PEC calculation

Own address register

Dual address register

PEC register

Clock control

Control registers (CR1&CR2)

Status registers (SR1&SR2)

Control logic

Interrupts

DMA requests & ACK

SDA

SCL

SMBA
I²C Slave Mode

- By default I²C operates in slave mode
- Detect start condition
- Compare the address received from the SDA line with interface address (OAR1) and with OAR2 (if ENDUAL=1) or the General Call address (if ENGC=1)
  - Header or address not matched: ignores it and wait for another start condition
  - Header matched (10-bit mode only)
  - Address matched: ACK or Interrupt or read the DUALF bit

Following the address reception and after clearing ADDR Slave will either
  - Send bytes from DR register to SDA line via shift register if in transmitter mode
  - Receive bytes from SDA line to DR register via shift register if in receiver mode

- ACK or interrupt upon after each byte is completed.
I²C Master Mode

- Generation of a Start condition will switch the Salve mode to Master mode.
- Master initiates a data transfer and generates the clock signal.
- After the Start condition, read the Status register 1 and write the Data register with the Slave address.
- Then wait for a read of the SR1 register followed by a read of the SR2 register.
- Then send the slave address with LSB cleared to enter Transmitter mode or with LSB set to enter Receiver mode.
- Following the address reception and after clearing ADDR Master will either:
  - Send bytes from DR register to SDA line via shift register if in transmitter mode.
  - Receive bytes from SDA line to DR register via shift register if in receiver mode.
- Wait for ACK or ACK and then continue for the next byte.
- Send Stop condition.
I²C Control Register 1 I2C_CR1

- **ACK**: Acknowledge enable
- **STOP**: Stop generation
  - In master mode, if set, will generate a Stop condition after the current byte transfer
  - In slave mode, if set, will release the SCL and SDA line after the current byte transfer
- **START**: Start generation
- **PE**: Peripheral enable

- When STOP, START or PEC bit is set, the software must not perform any write access to CR1 before this bit is cleared by hardware in case it does not set a second STOP, START or PEC request.
- Must not reset PE bit before the end of the communication in master mode.
# I²C Control Register 2 I2C_CR2

<table>
<thead>
<tr>
<th>Bit 15</th>
<th>Bit 14</th>
<th>Bit 13</th>
<th>Bit 12</th>
<th>Bit 11</th>
<th>Bit 10</th>
<th>Bit 9</th>
<th>Bit 8</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>LAST</td>
<td>DMA EN</td>
<td>ITBUF EN</td>
<td>ITEVT EN</td>
<td>ITERR EN</td>
<td>Reserved</td>
<td>FREQ[5:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
</tr>
</tbody>
</table>

- **Interrupt enable bits**
- **FREQ[5:0]: Peripheral clock frequency**
  - I²C peripheral connected to APB
  - Configure the clock frequency between 2MHz(0b000010) to 42MHz(0b101010)
  - Baud rate = Freq (if not in fast mode)
### I²C Own address registers

<table>
<thead>
<tr>
<th>Address Mode</th>
<th>OAR1</th>
<th>OAR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD[14:8]</td>
<td>rw</td>
<td>rw</td>
</tr>
<tr>
<td>ADD[7:1]</td>
<td>rw</td>
<td>rw</td>
</tr>
<tr>
<td>ADD0</td>
<td>rw</td>
<td>rw</td>
</tr>
</tbody>
</table>

#### OAR1
- **ADDMODE**: 7-bit slave address or 10-bit slave address
- Bit 14 should be kept at 1 by software
- 10-bit mode: ADD[9:0]: interface address
- 7-bit mode: ADD[7:1]: interface address

#### OAR2
- Dual addressing mode
- ENDUAL: Enable dual addressing mode with 1
- ADD2[7:1]: secondary address in dual addressing mode
I²C Status Register 1 I2C_SR1

- Error related bits
  - TxE: Data register empty (transmitters)
    - 0: DR not empty
    - 1: DR empty
  - RxNE: Date register empty (receivers)
    - 0: DR empty
    - 1: DR not empty
  - STOPF: Stop detection (Slave mode)
  - BTF: Byte transfer finished
  - ADDR: Address sent (master mode)/matched(slave mode)
  - SB: Start bit(Master mode)
## I²C Status Register 2 I2C_SR2

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>GENCALL: General call address (slave mode)</td>
</tr>
<tr>
<td>6</td>
<td>TRA: Transmitter / Receiver</td>
</tr>
<tr>
<td>5</td>
<td>BUSY: Bus busy</td>
</tr>
<tr>
<td>4</td>
<td>MSL: Master / Slave</td>
</tr>
<tr>
<td>3</td>
<td>Res.</td>
</tr>
<tr>
<td>2</td>
<td>TRA</td>
</tr>
<tr>
<td>1</td>
<td>BUSY</td>
</tr>
<tr>
<td>0</td>
<td>MSL</td>
</tr>
</tbody>
</table>

### Field Descriptions
- **PEC[7:0]**: Parity Error Check
- **DUALF**: Dual Mode
- **SMB HOST**: SMB Host
- **SMBDE FAULT**: SMB Device Fault
- **GENCALL**: General Call
- **BUSY**: Bus Busy
- **MSL**: Master / Slave
Required sequence

- Program the peripheral input clock in I2C_CR2 Register to generate right timings
- Configure the clock control registers
- Configure the rise time registers
- Enable the peripheral
- Set the START bit to generate a Start condition
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

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Speed</th>
<th>Signals Req. for Bidirectional Communication with N devices</th>
<th>Device Addressing</th>
<th>Topology</th>
</tr>
</thead>
<tbody>
<tr>
<td>UART (Point to Point)</td>
<td>Fast – Tens of Mbit/s</td>
<td>2*N (TxD, RxD)</td>
<td>None</td>
<td>Point-to-point full duplex</td>
</tr>
<tr>
<td>UART (Multi-drop)</td>
<td>Fast – Tens of Mbit/s</td>
<td>2 (TxD, RxD)</td>
<td>Added by user in software</td>
<td>Multi-drop</td>
</tr>
<tr>
<td>SPI</td>
<td>Fast – Tens of Mbit/s</td>
<td>3+N for SCLK, MOSI, MISO, and one SS per device</td>
<td>Hardware chip select signal per device</td>
<td>Multi-point full-duplex, multi-drop half-duplex buses</td>
</tr>
<tr>
<td>I²C</td>
<td>Moderate – 100 kbit/s, 400 kbit/s, 1 Mbit/s, 3.4 Mbit/s. Packet overhead.</td>
<td>2: SCL, SDA</td>
<td>In packet</td>
<td>Multi-point half-duplex bus</td>
</tr>
</tbody>
</table>