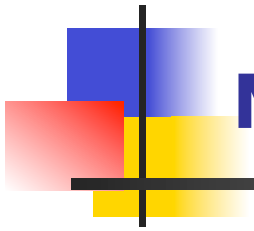


Microprocessor Systems

Serial Communications

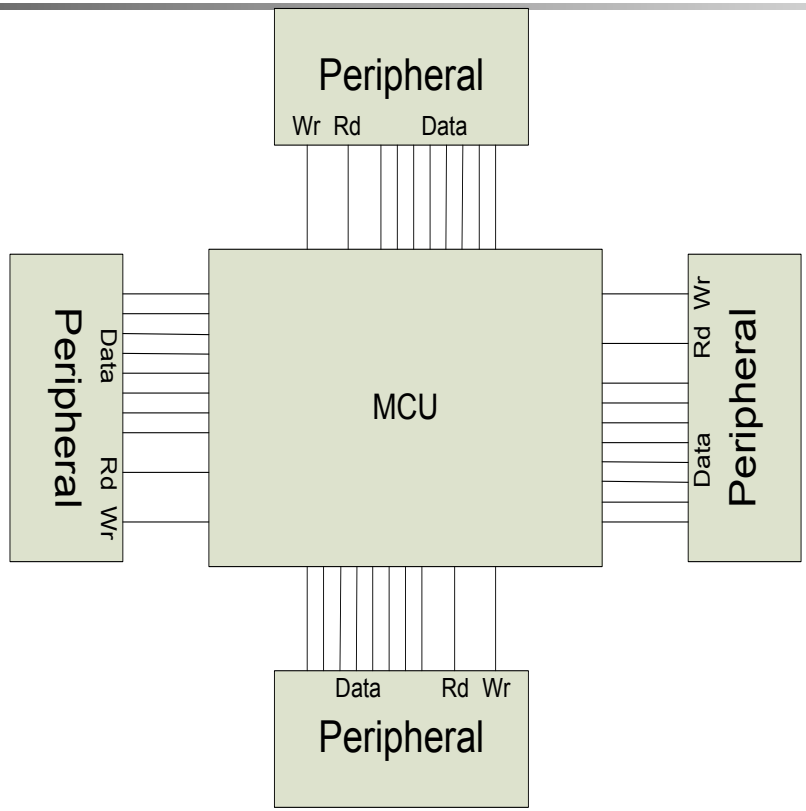
Overview

- Serial communications
  - Concepts
  - Tools
  - Software: polling, interrupts and buffering
- UART communications
  - Concepts
  - KL25 I2C peripheral
- Serial Peripheral Interface communications
  - Concepts
  - KL25 SPI peripheral

Why Communicate Serially?

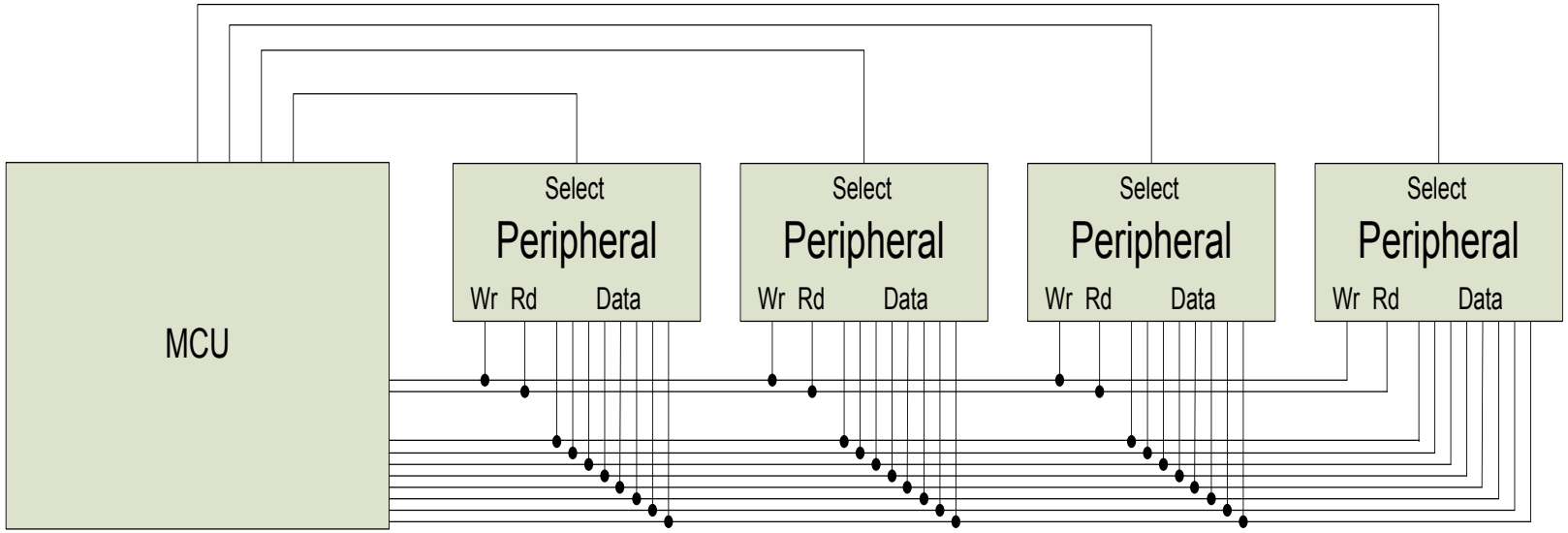
- Although native word size for CPU is 32 bits, sending all of a word's bits simultaneously has disadvantages:
  - **Cost and weight:** larger IC package, more wires, larger connectors
  - **Mechanical reliability:** more wires => more connector contacts to fail
  - **Timing complexity:** some bits may 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

Example System



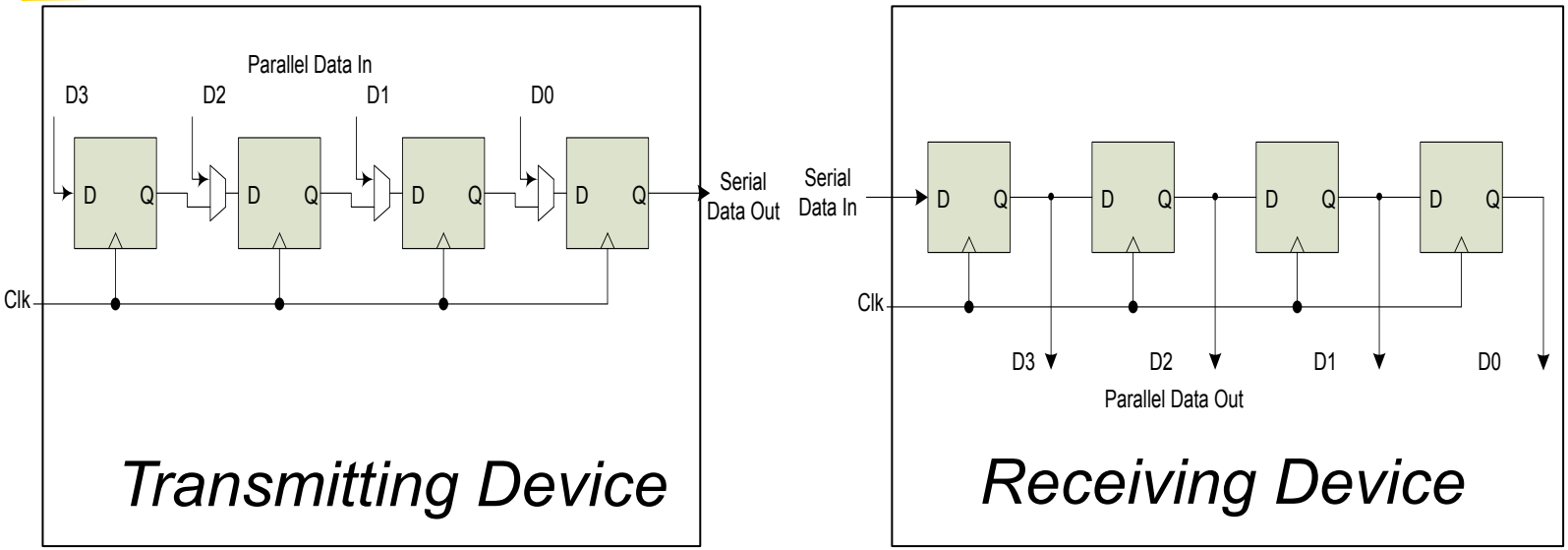
- Dedicated point-to-point connections
  - Parallel data lines, read and write lines between MCU and each peripheral
- Fast, allows simultaneous transfers
- Requires many connections, Printed Circuit Board (PCB) area, scales badly
  - Need 4*(8+2) = 40 pins on MCU to communicate!

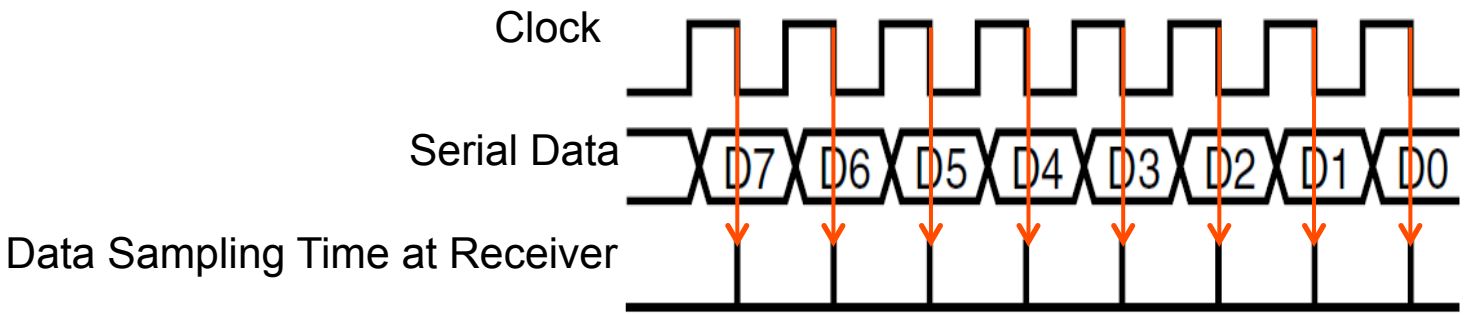
Parallel Buses



- 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

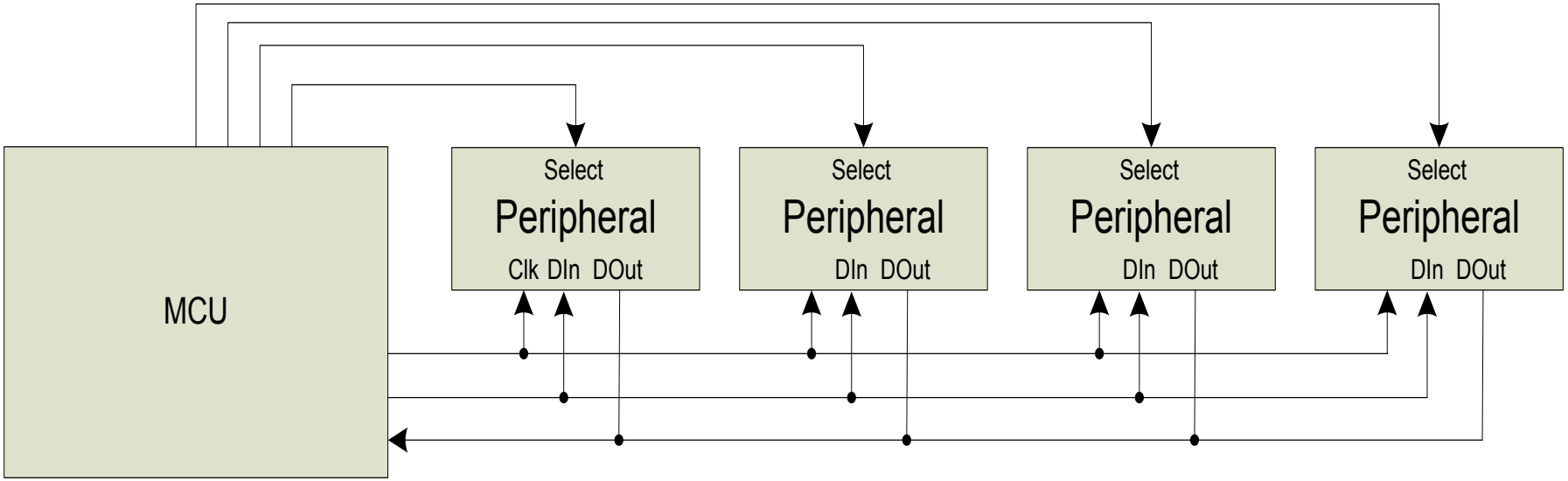
Synchronous Serial Data Transmission





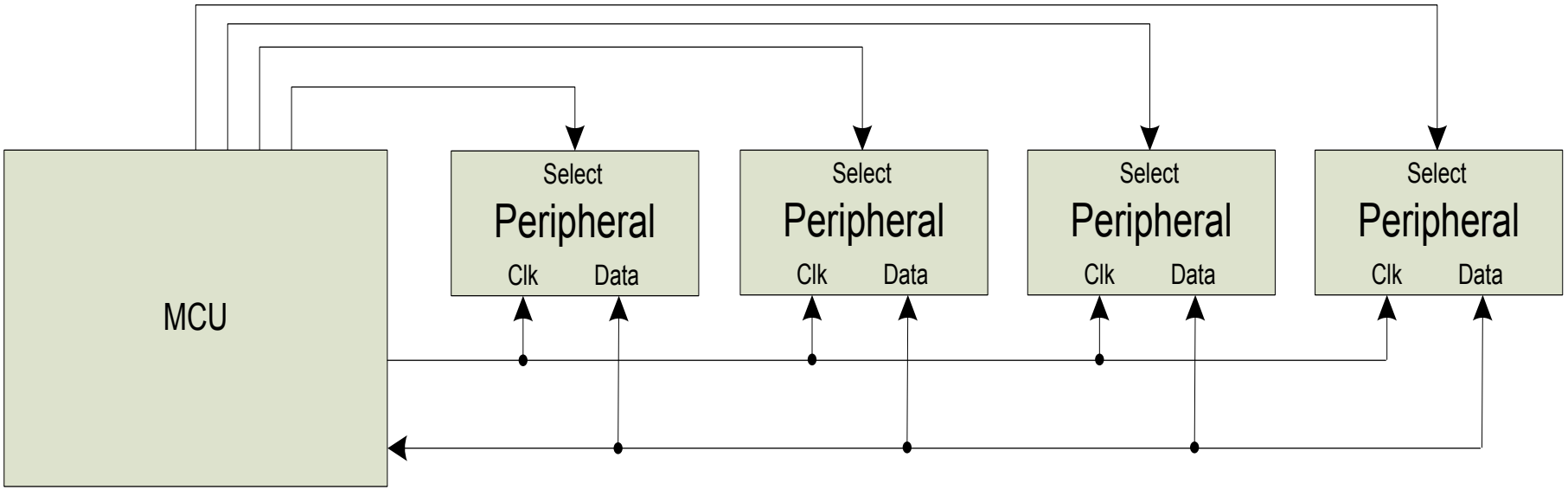
- Use shift registers and a clock signal to convert between serial and parallel formats
- Synchronous: an explicit clock signal is along with the data signal

Synchronous Full-Duplex Serial Data Bus



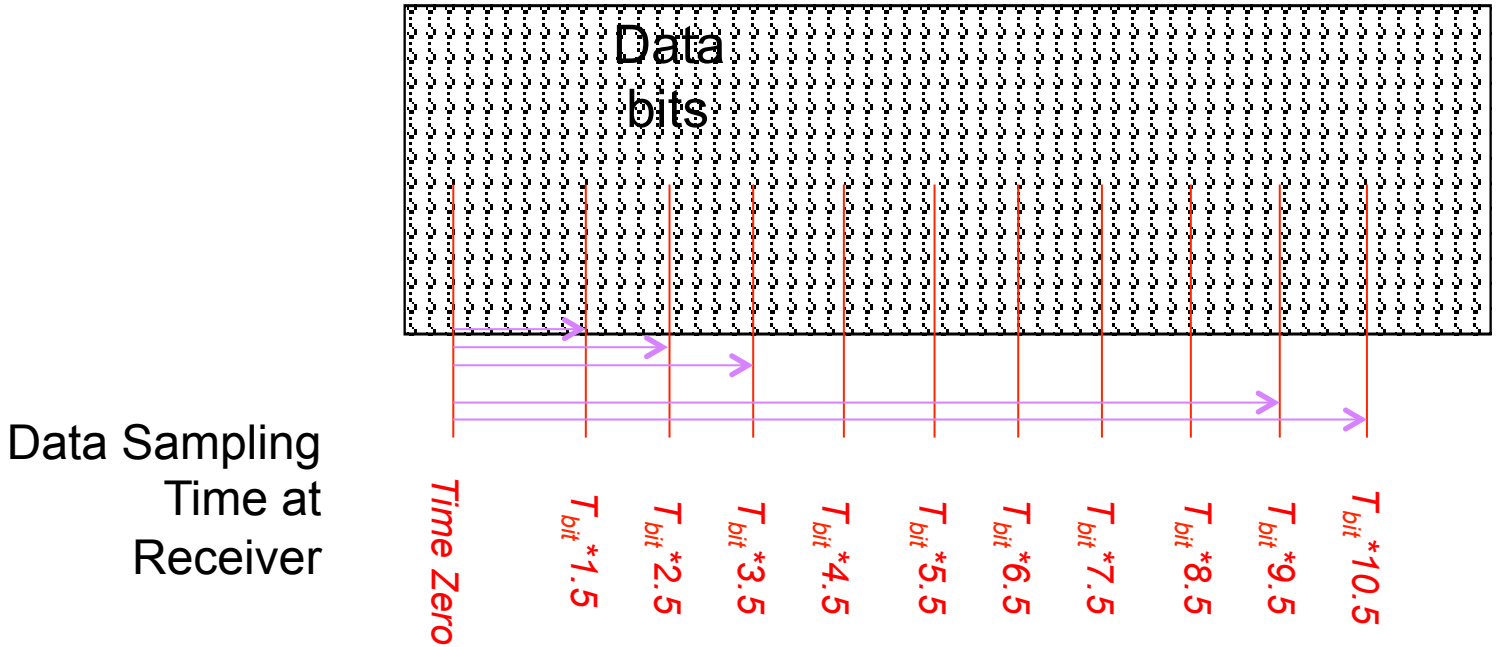
- Now can use two serial data lines - one for reading, one for writing.
  - Allows simultaneous send and receive *full-duplex communication*
  - Need 4 + 3 = 7 pins on MCU to communicate

Synchronous Half-Duplex Serial Data Bus



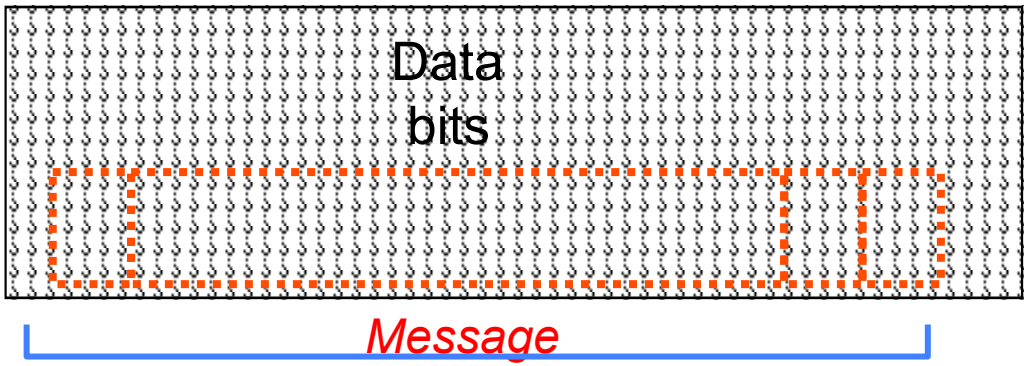
- **Share the serial data line**
  - Need 4 + 2 = 6 pins on MCU to communicate
- **Doesn't allow simultaneous send and receive - is *half-duplex communication***

Asynchronous Serial Communication



- 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_{bit}*(N +1.5)
- Stop bit is also used to detect some timing errors

Serial Communication Specifics

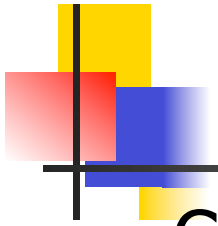


- Data frame fields
  - Start bit (one bit)
  - Data (LSB first or MSB, and size – 7, 8, 9 bits)
  - Optional parity bit is used to make total number of ones in data even or odd
  - Stop bit (one or two bits)
- All devices must use the same communications parameters
  - E.g. communication speed (300 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.

SOFTWARE STRUCTURE – HANDLING ASYNCHRONOUS COMMUNICATION

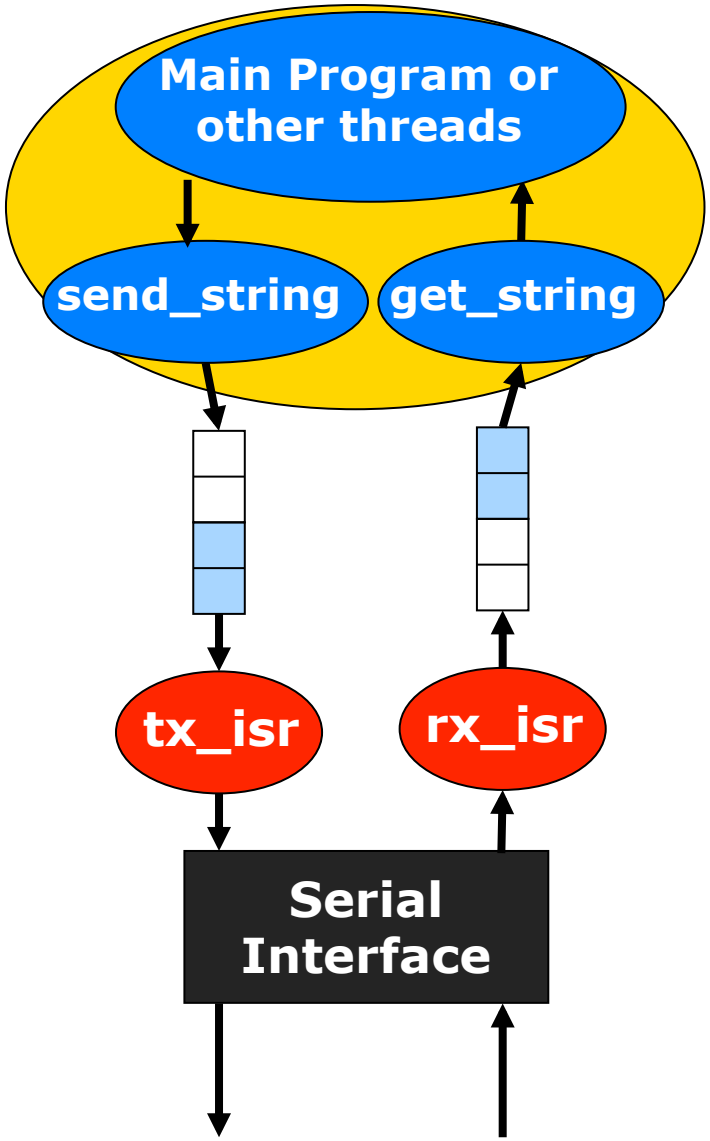


Software Structure

- 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 use 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

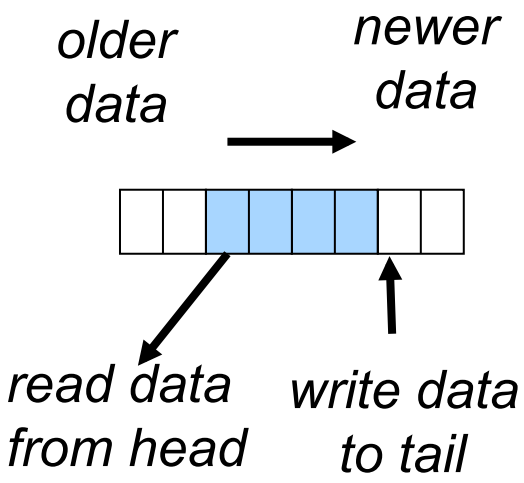
- ARM Cortex-M0+ provides one IRQ for all of a communication interface's events
- Within ISR (IRQ Handler), need to determine what triggered the interrupt, and then service it

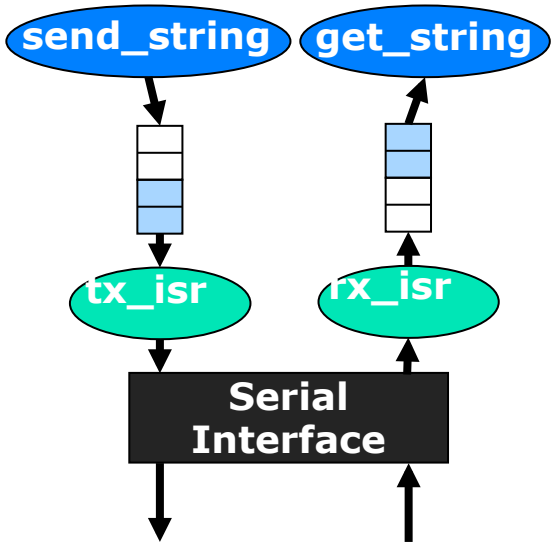
```
void UART2_IRQHandler() {
   if (transmitter ready) {
       if (more data to send) {
              get next byte
              send it out transmitter
       }
   }
   if (received data) {
       get byte from receiver
       save it
   }
   if (error occurred) {
       handle error
   }

}
```

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
  - 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 the 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 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

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]`

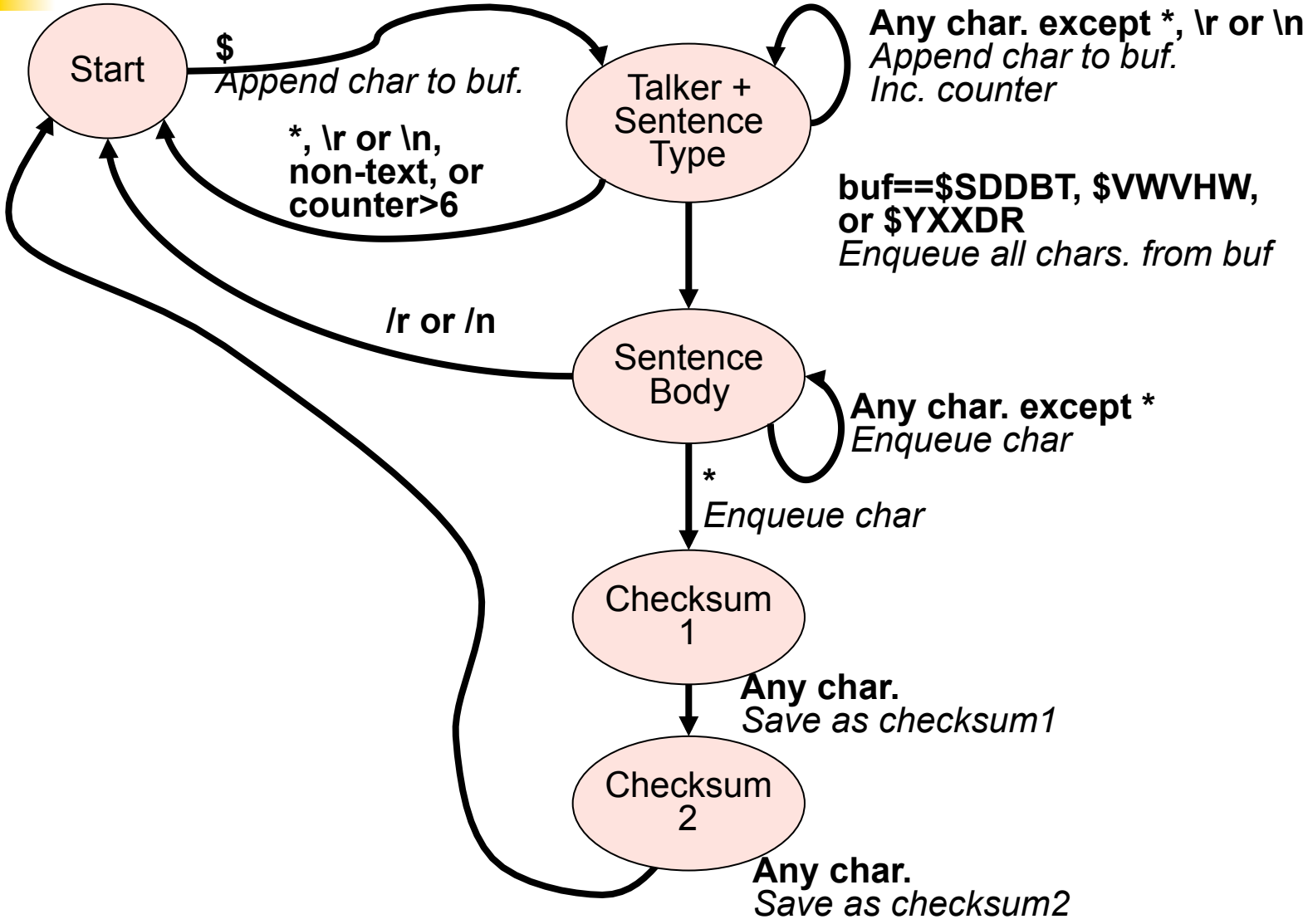
"$"	The "$" signifies the start of a message.
ID	The talker identification is a two letter mnemonic which describes the source of the navigation information. The GP identification signifies a GPS source.
MSG	The message identification is a three letter mnemonic which describes the message content and 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

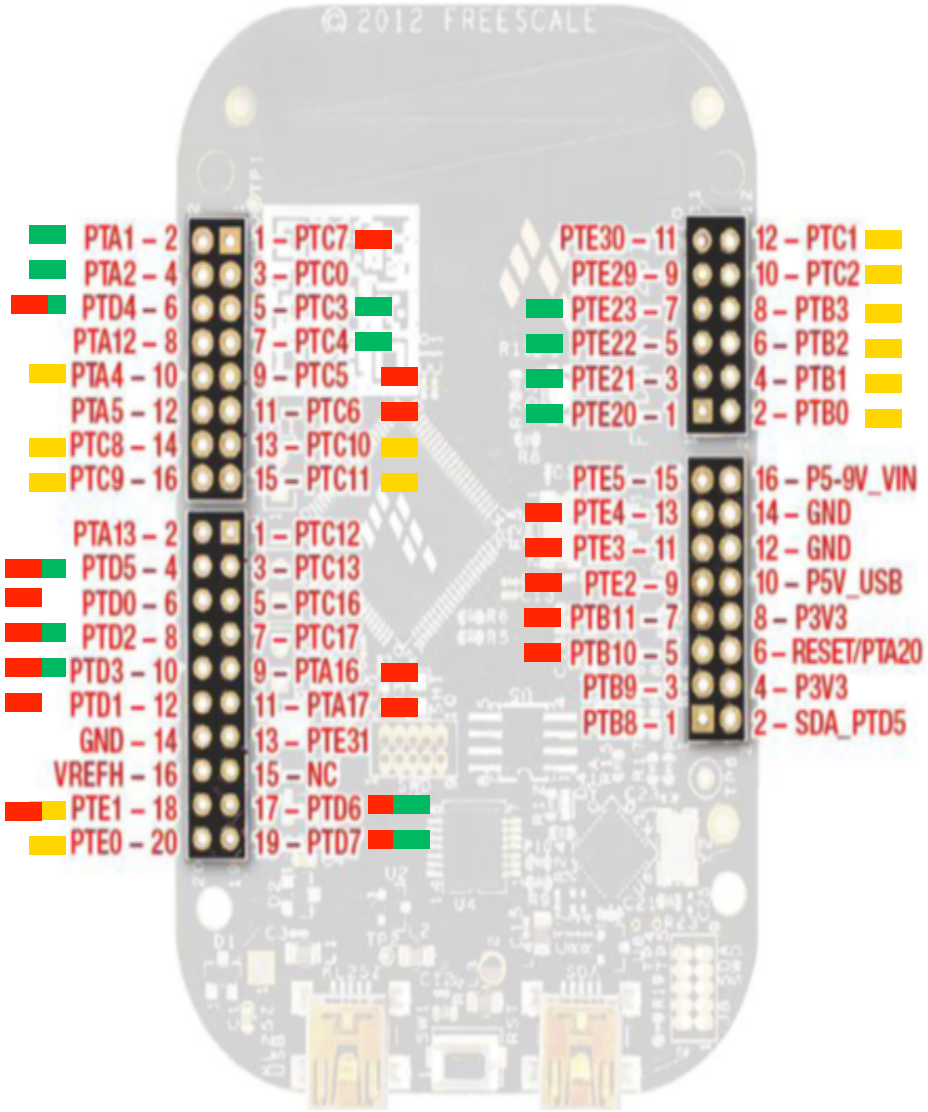


Parsing

```
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

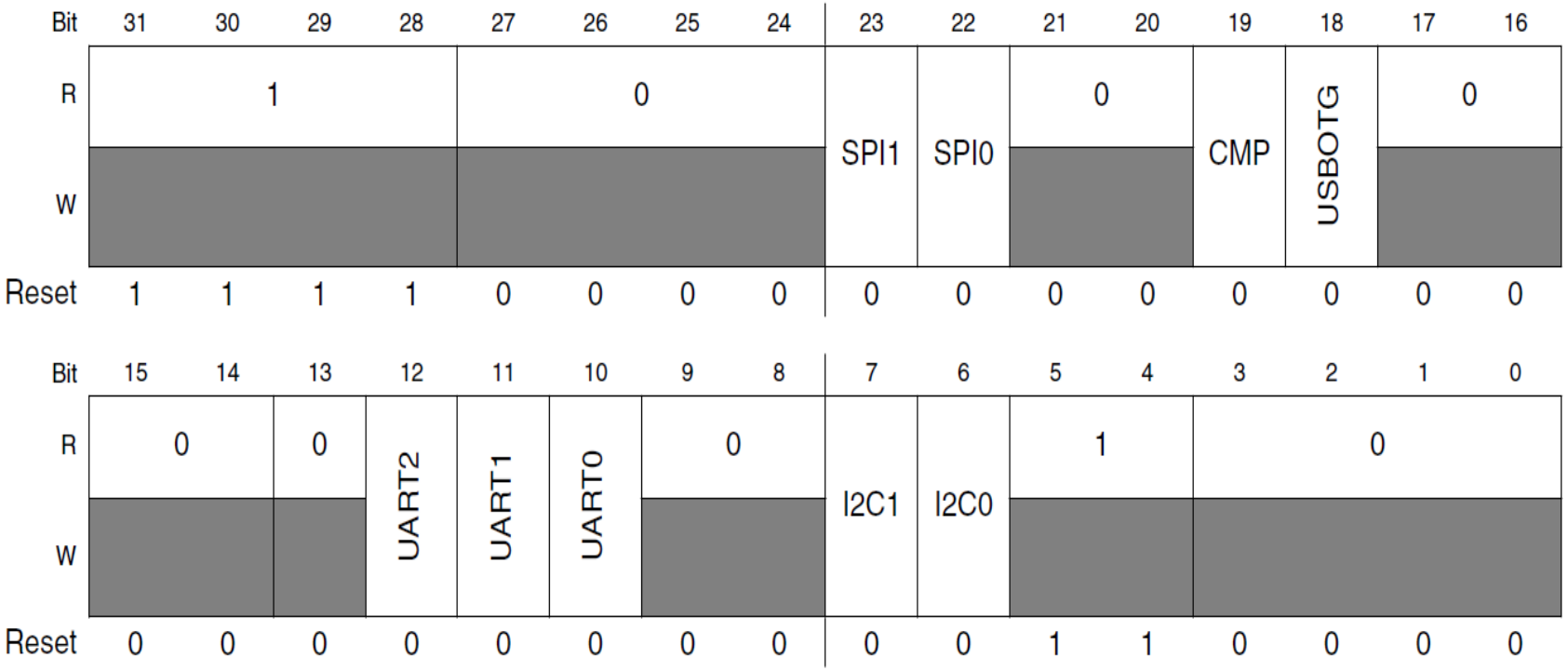


UART

SPI

I^2C

KL25Z Clock Gating for Serial Comm.



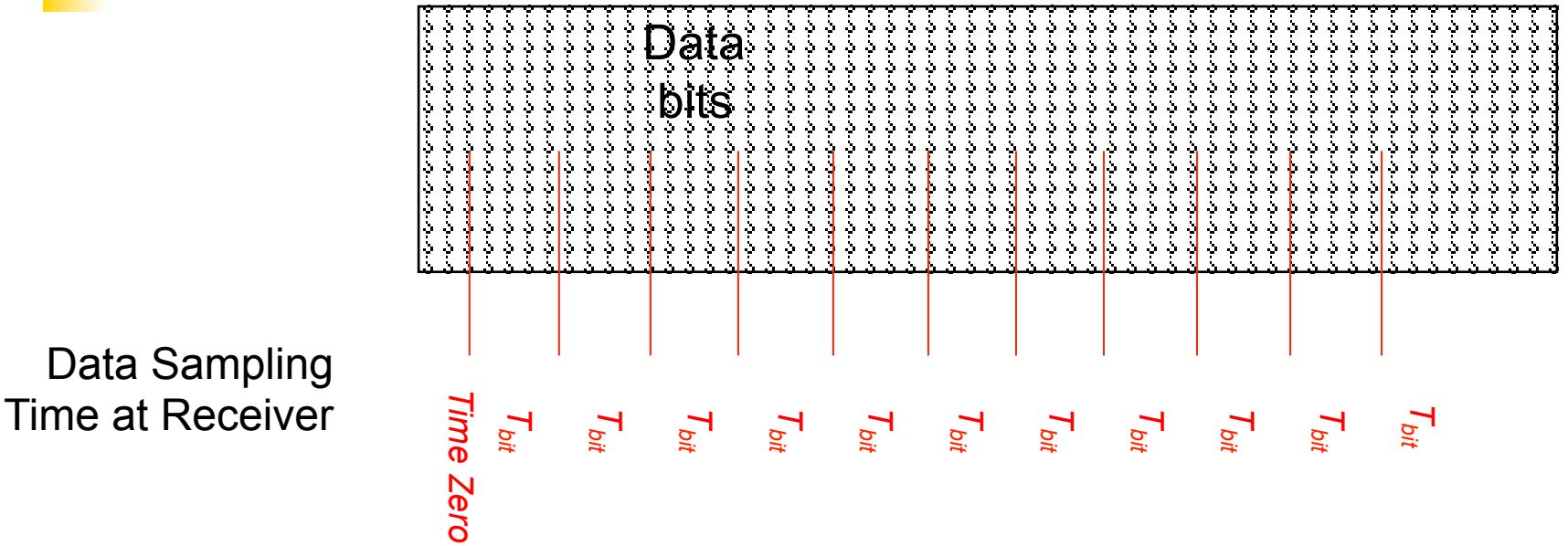
Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	1				0				SPI1	SPI0	0		CMP	USBOTG	0	
W																
Reset	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0

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

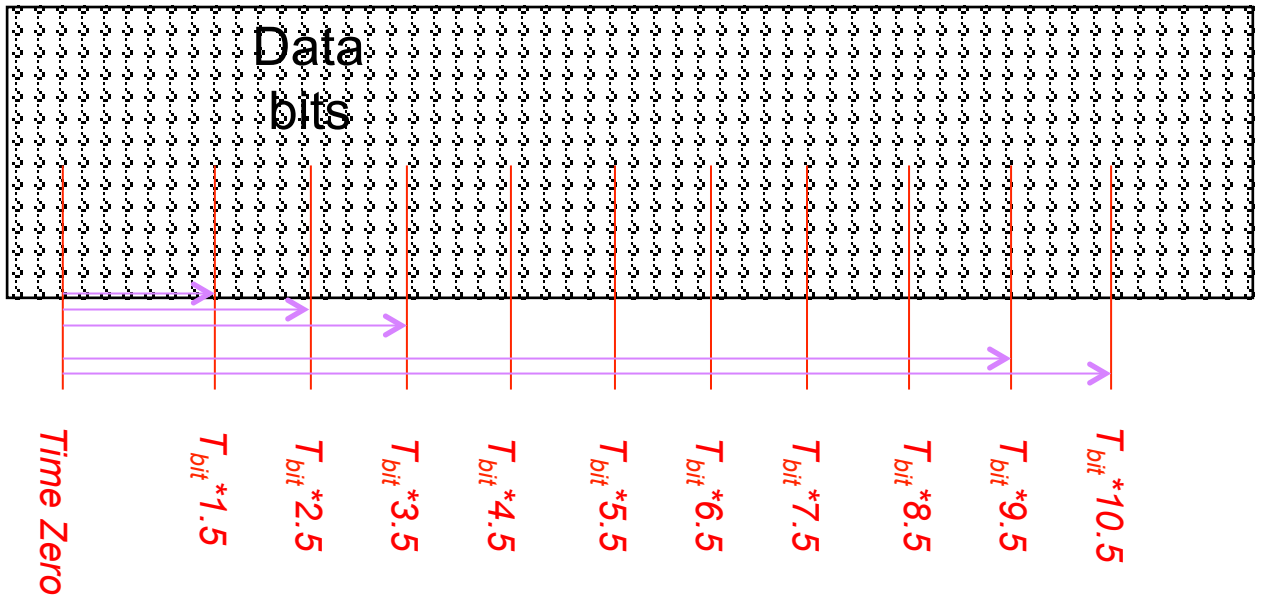
ASYNCHRONOUS SERIAL (UART) COMMUNICATIONS

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!

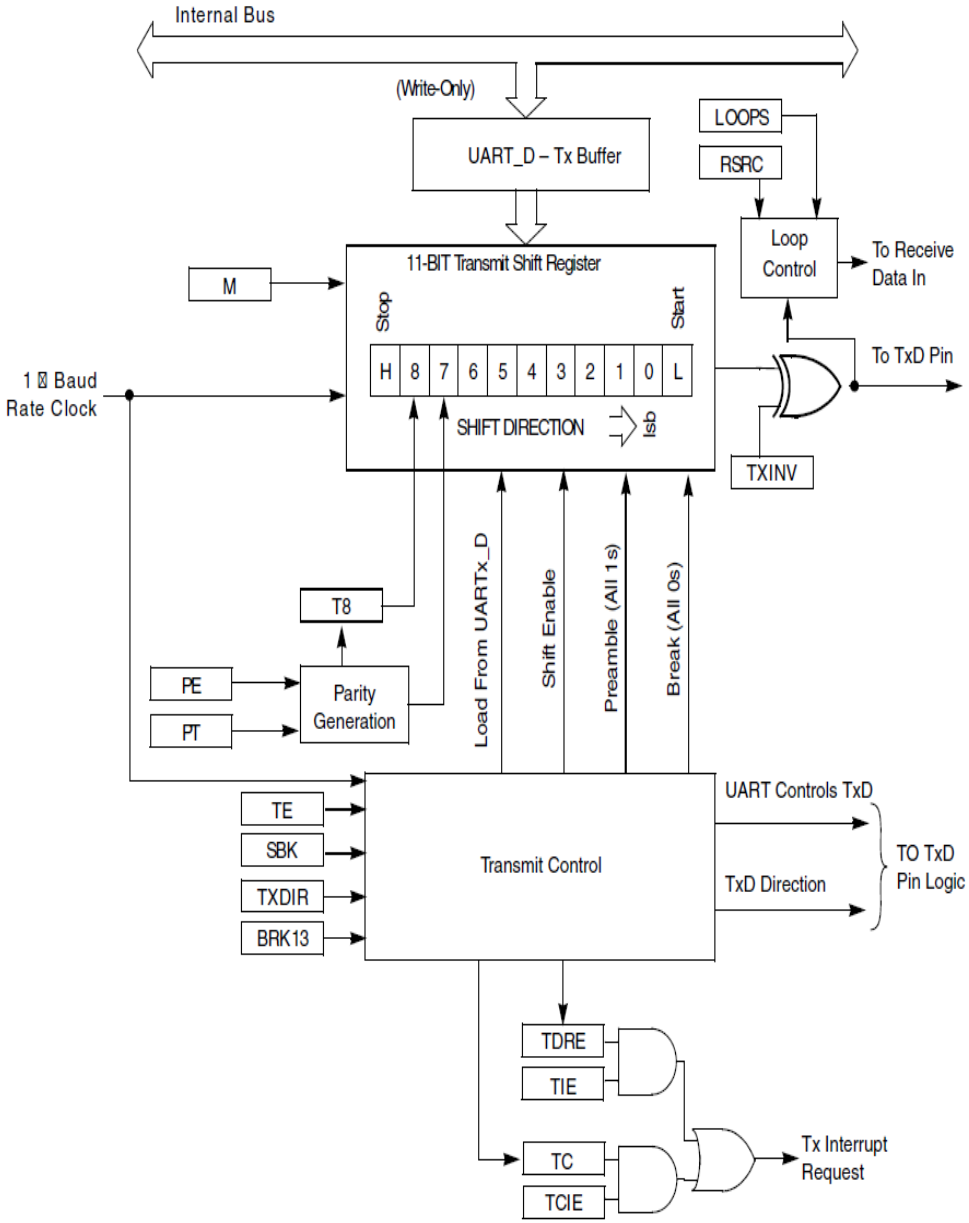
For this to work…

- 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

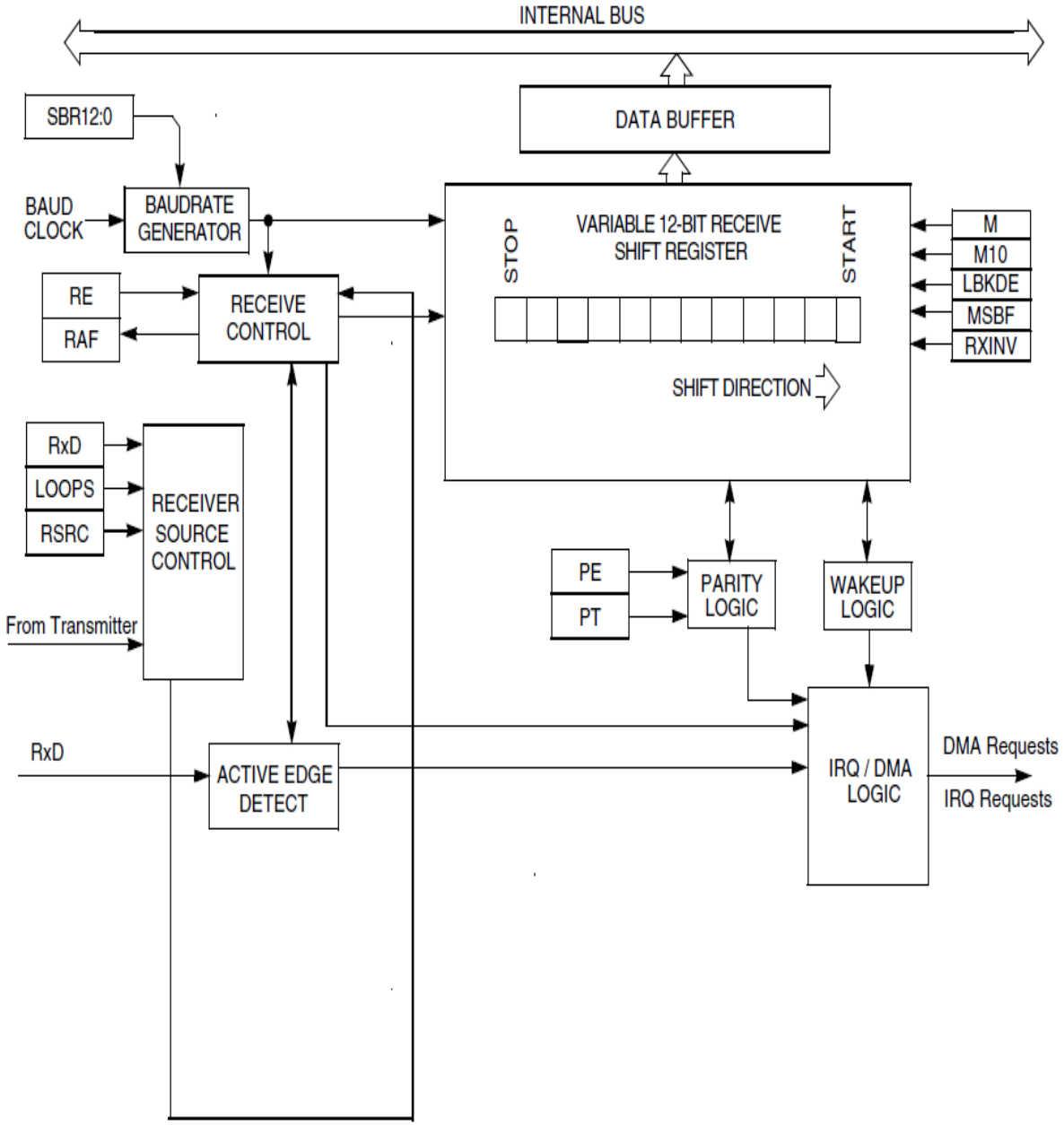
KL25 UARTs

- 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

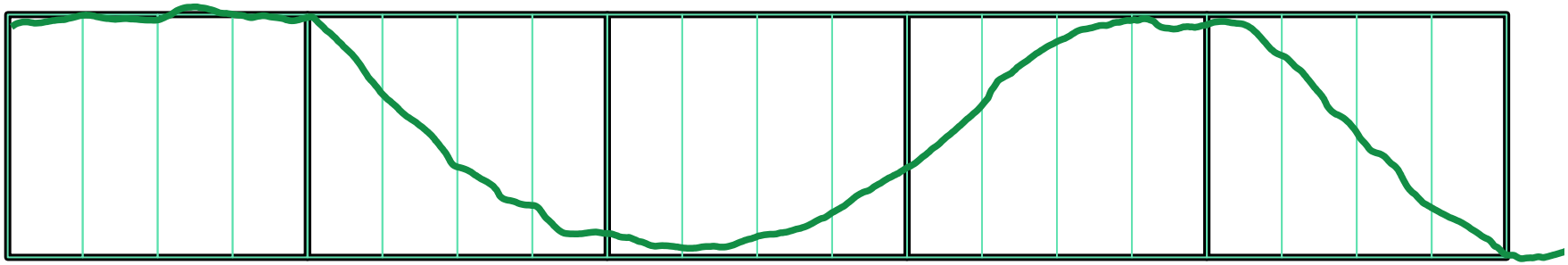
UART Transmitter



UART Receiver

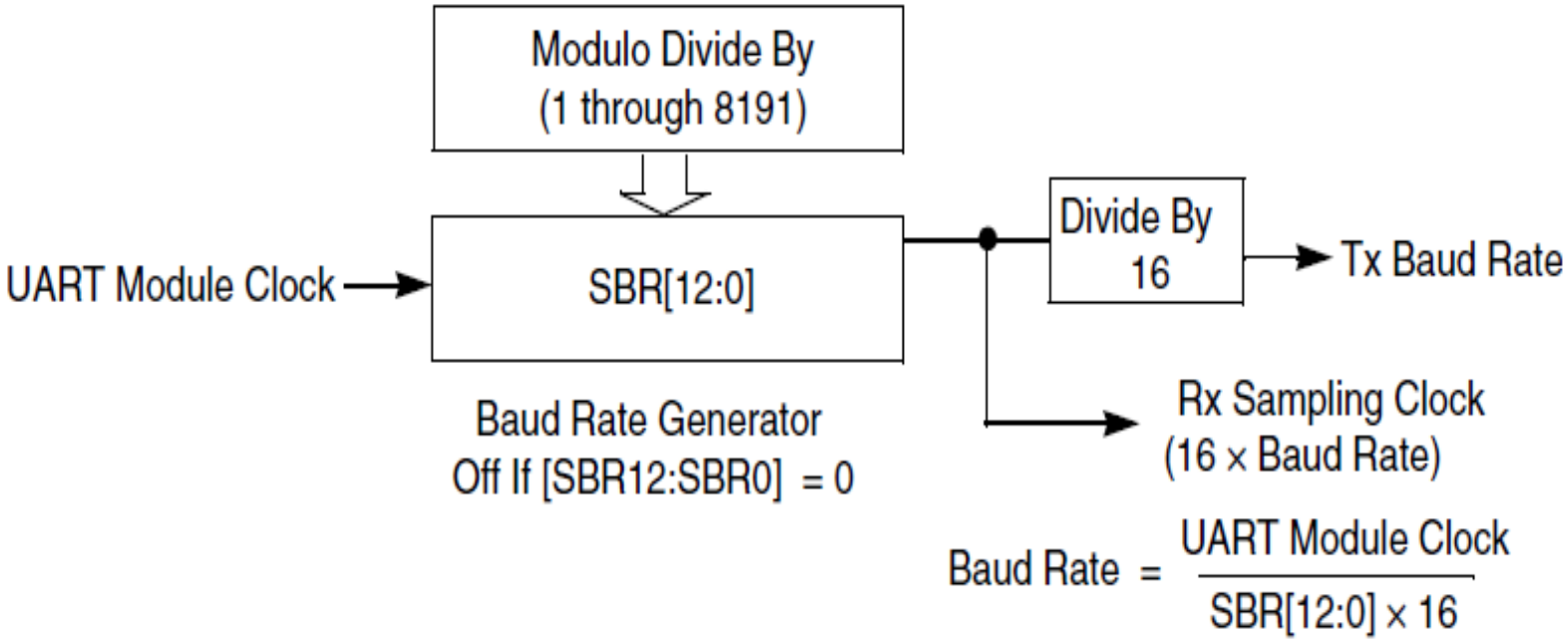


Input Data Oversampling



- 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 in with CMSIS)
- 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 in with CMSIS)

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
- DOZEN: 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 E
        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;
}
```

Software for Polled Serial Comm.

```
void UART2_Transmit_Poll(uint8_t data) {
      // wait until transmit data register is empty
      while (!(UART2->S1 & UART_S1_TDRE_MASK))
            ;
      UART2->D = data;
}

uint8_t UART2_Receive_Poll(void) {
      // wait until receive data register is full
      while (!(UART2->S1 & UART_S1_RDRF_MASK))
            ;
      return UART2->D;
}
```

Example Transmitter

```
while (1) {
      for (c='a'; c<='z'; c++) {
            UART2_Transmit_Poll(c);
}
```

Example Receiver: Display Data on LCD

```
line = col = 0;
while (1) {
        c = UART2_Receive_Poll();
        Set_Cursor(col, line);
        lcd_putchar(c);
        col++;
        if (col>7) {
                        col = 0;
                        line++;
                        if (line > 1) {
                                line = 0; }
        }
}
```

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, 128);
        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

            }
}
```