Communication Devices Structure and Programming

Overview

- **Serial communications**
  - Concepts
  - Tools
  - Software:
    - polling,
    - interrupts and buffering
- **UART communications**
  - Concepts
  - KL25 I2C peripheral
- **DMA – Direct Memory Access**
  - Concepts
  - KL DMAMUX and DMA module

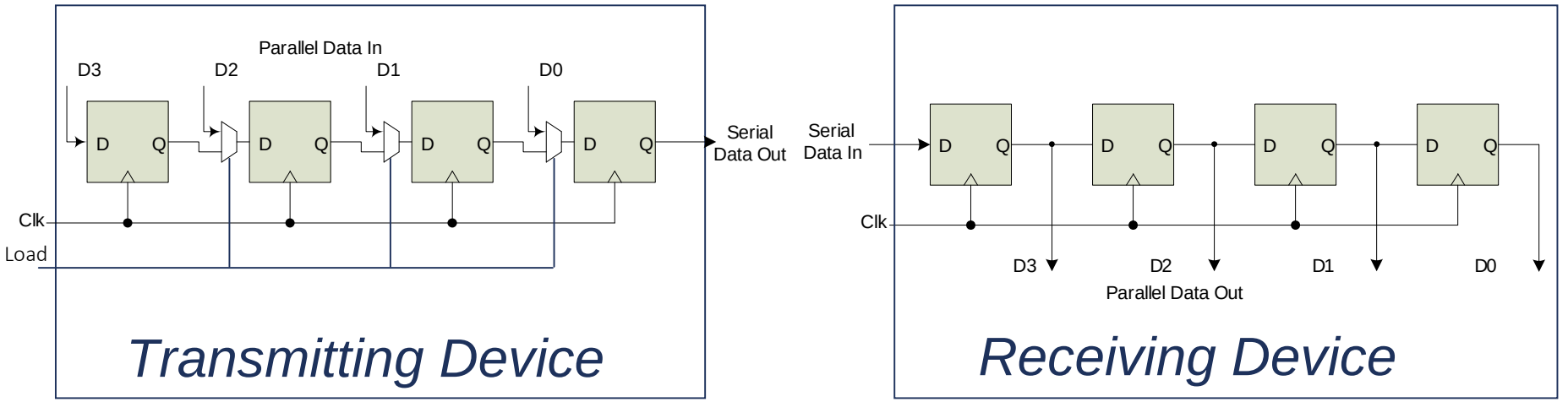
SERIAL COMMUNICATION

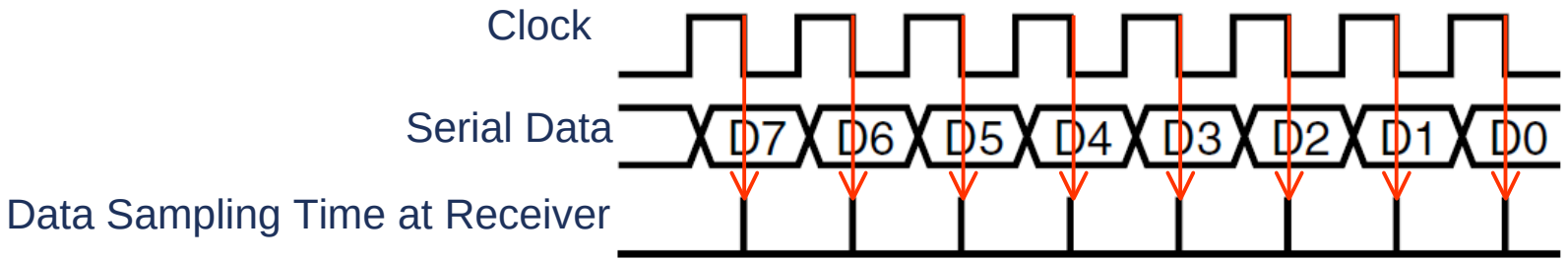
Why Communicate Serially?

- **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
- **Serial Communication examples:**
  - UART, SPI, I2C
  - SATA
  - PCIe

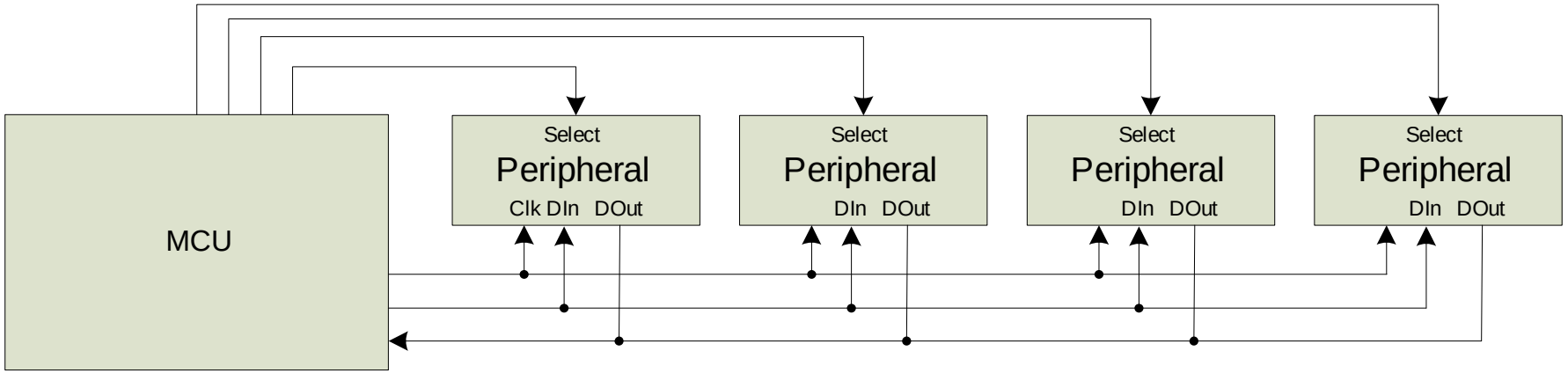
Synchronous Serial Data Transmission





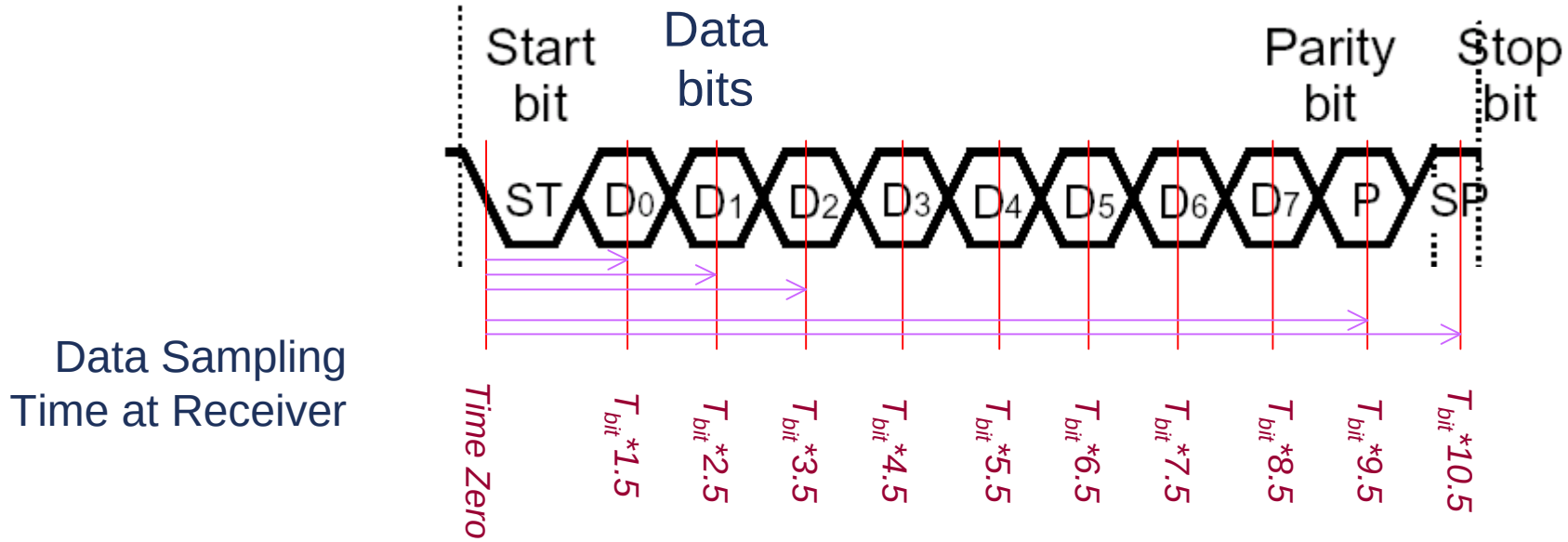
- **Use shift registers and a clock signal to convert between serial and parallel formats**
- **Synchronous: an explicit clock signal is along with the data signal**

Synchronous Full-Duplex Serial Data Bus



- **Solves a problem of two-way communication**
- **We can use two separate serial data lines - one for reading, one for writing.**
  - Allows simultaneous send and receive: **full-duplex communication**

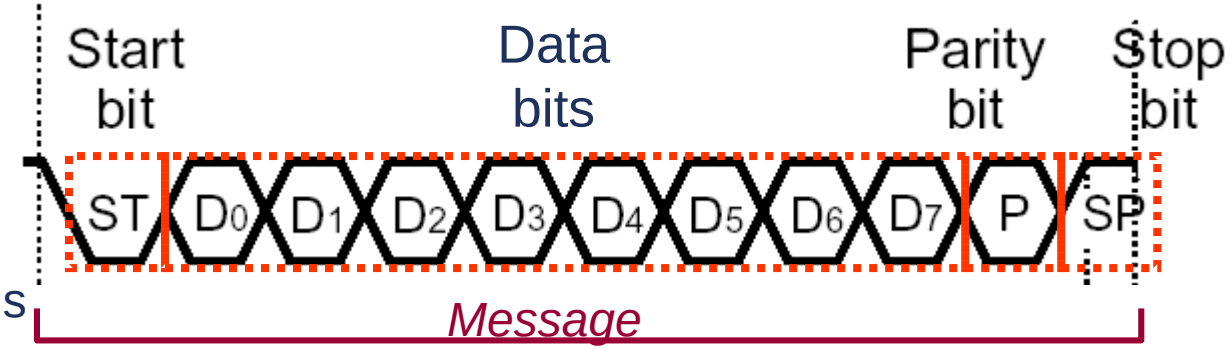
Asynchronous Serial Communication



- **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)

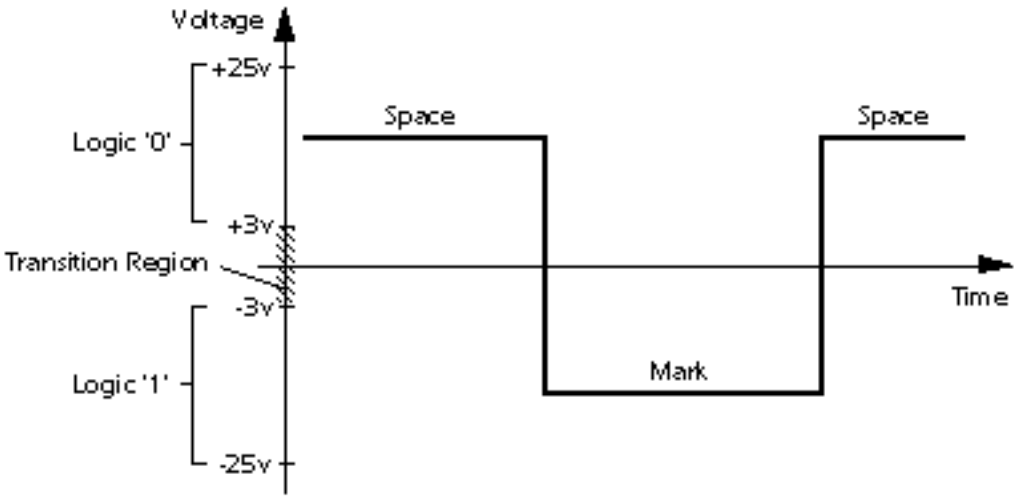


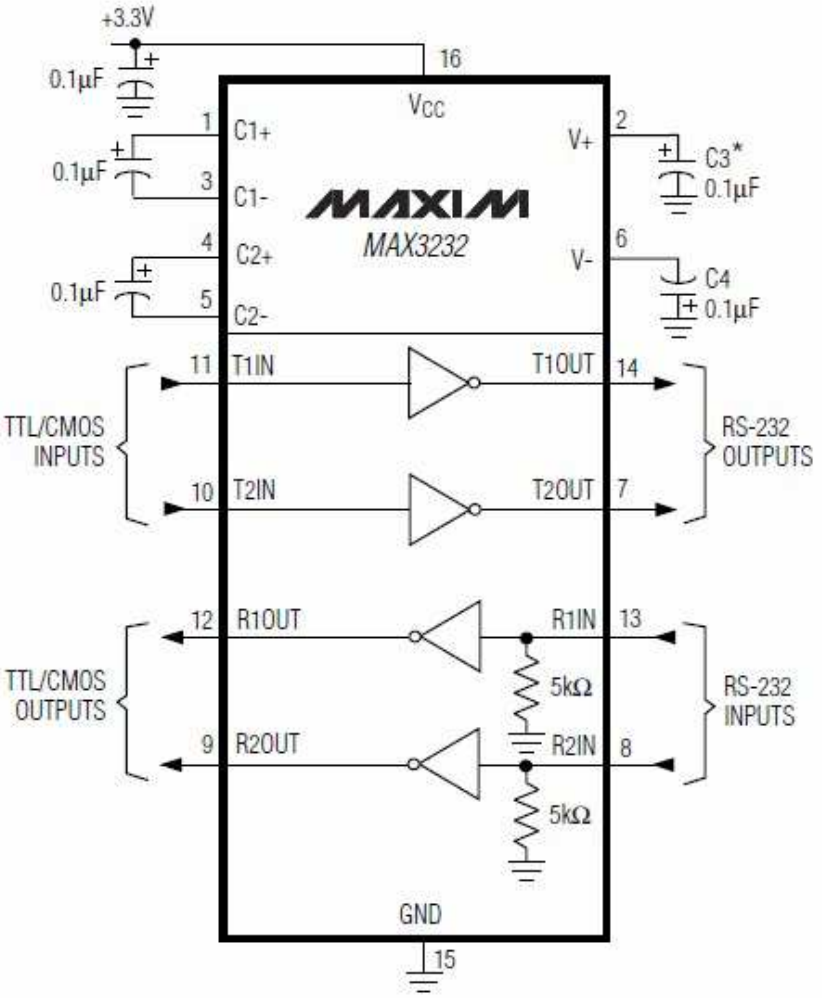
- **All devices must use the same communications parameters**
  - E.g. communication speed: **baud rate** (300, 600, 1200, 2400, 9600, 14400, **19200**, 38400, 57600, 115200 bits/second)
- **Sophisticated network protocols have more information in each data frame**
  - Medium access control – when multiple 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. CRC - 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.

Solution to Noise: Higher Voltages



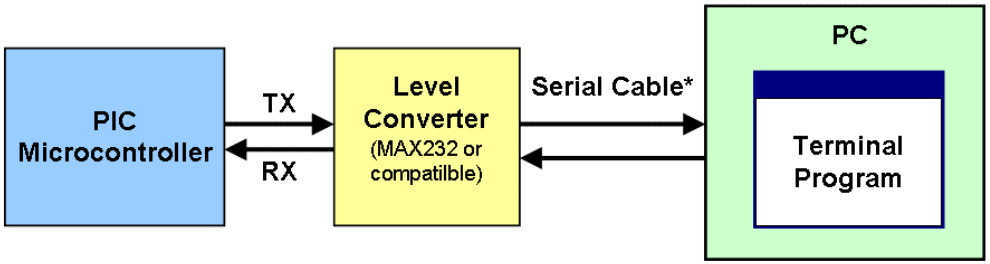


- **Use higher voltages to improve noise margin:**
  +3 to +15 V, -3 to -15 V

- **Example IC (Maxim MAX3232) uses charge pumps to generate higher voltages from 3.3V supply rail**

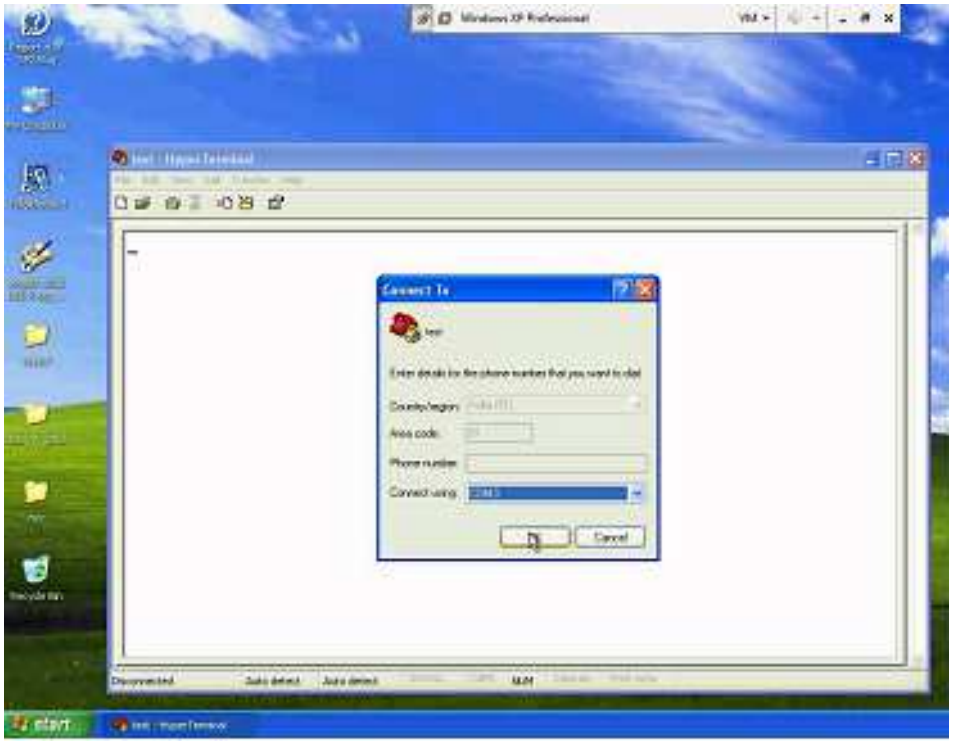
ASCII communication with PC

- **Today USB-Serial converter is used to install COM device on a PC**
- **ASCII codes send as data by a microcontroller to PC**
  - One data frame is one character
- **ASCII data can be viewed on any terminal software program**
  - e.g. HyperTerminal, Putty, etc.
- **Some programms can display data as binary also**
  - e.g. RealTerm

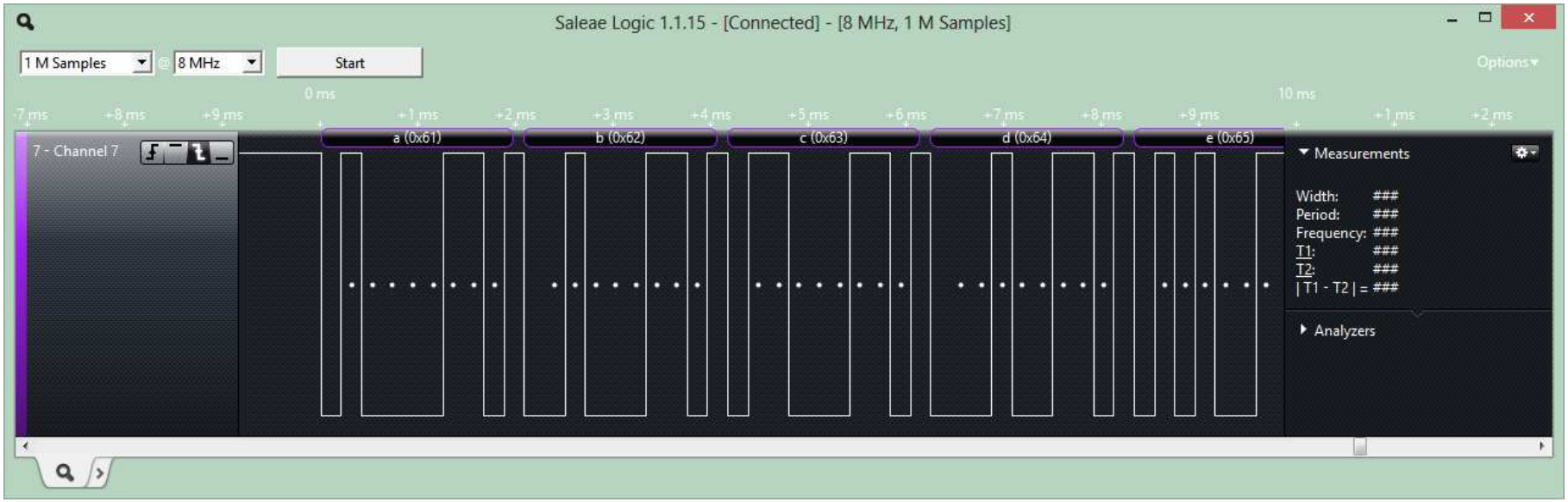


* or USB-to-Serial Converter

rs232_comm_block_diagram.ppt, v1.0,
Copyright © 22.10.2011 Christian Stadler



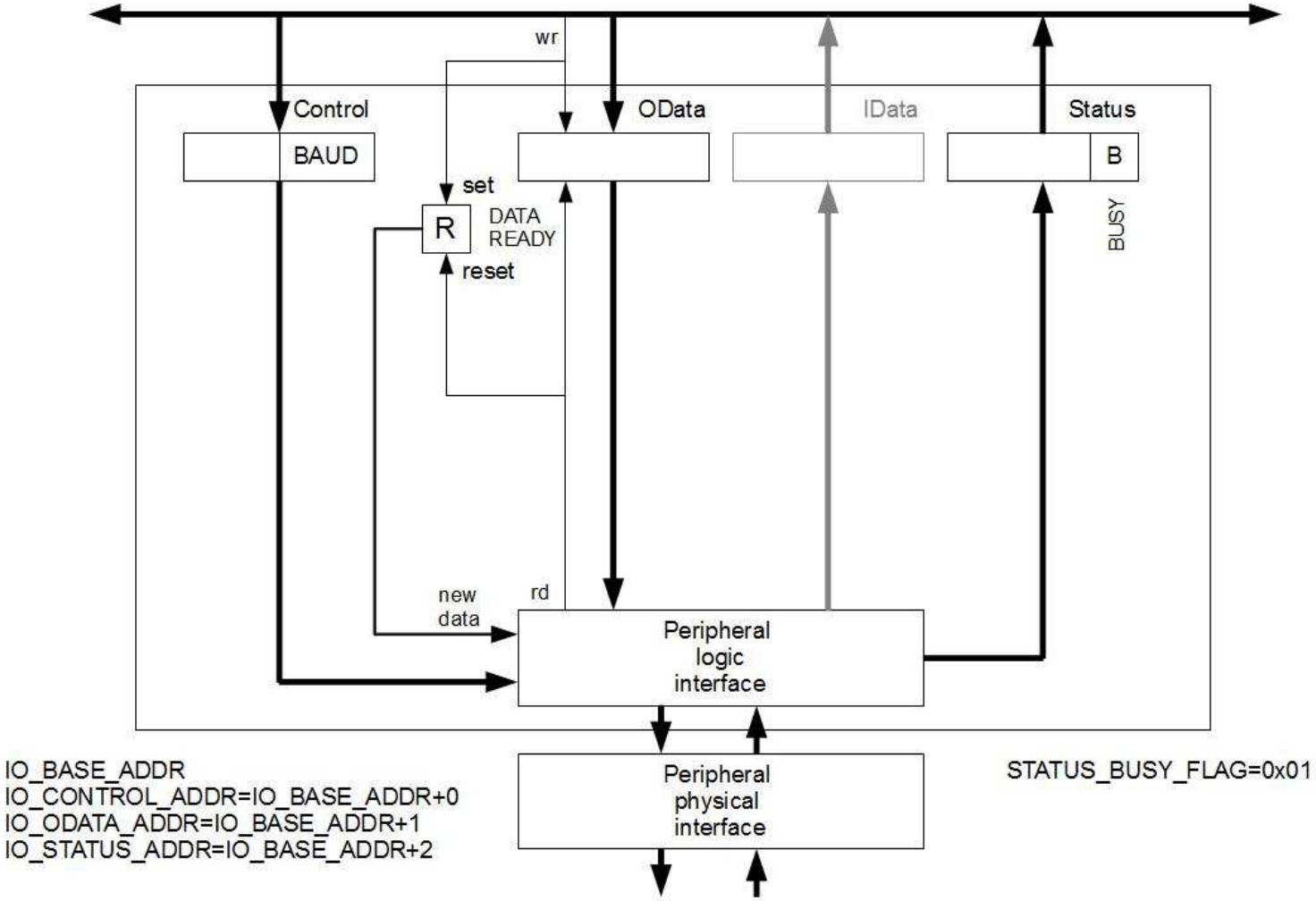
Tools for Serial Communications Development



- **Tedious and slow to debug serial protocols with just an oscilloscope**
  - Instead use a **logic analyzer** to decode bus traffic
- **Analog Discovery Logic Analyzer**
  - Plugs into PC's USB port
  - Decodes UART, SPI, I2C
- **Use terminal software to sniff serial communication data e.g. 'RealTerm'**

SIMPLE COMMUNICATION DEVICES

Simple low speed device



Simple device explained

- **The simple communication device occupies four locations in the system memory space (starts with IO_BASE_ADDR location)**
  - Registers are: Input Data, Output Data, Status, and Control
- **READY flag is used internally to recognize that a new data was sent by the processor**
  - READY is set when processor writes to Output Data (OData) register
  - READY is cleared when peripheral reads OData
  - READY flag is unreadable by the processor
- **As long as the peripheral is processing/sending data, the BUSY flag in Status register is active**
  - Processor can check Status register to suspend a new data
- **To check BUSY flag, a processor reads Status register and masks BUSY flag**
- **Additionally CONTROL register can be used to set the peripheral transmisson parameters (e.g. the baud rate)**

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

Controlling IO device

- **The process can periodically check the status of an I/O device to determine the need for service**
  - To check if device data register is empty/full
- **This method is called POLLING**
- **This is the simplest way for an I/O device programming.**
- **The processor is totally in control and does all the work.**
- **But processor may read the Status register many times, only to find that the device has not yet completed a comparatively slow I/O operation**
  - This leads to overhead in communication

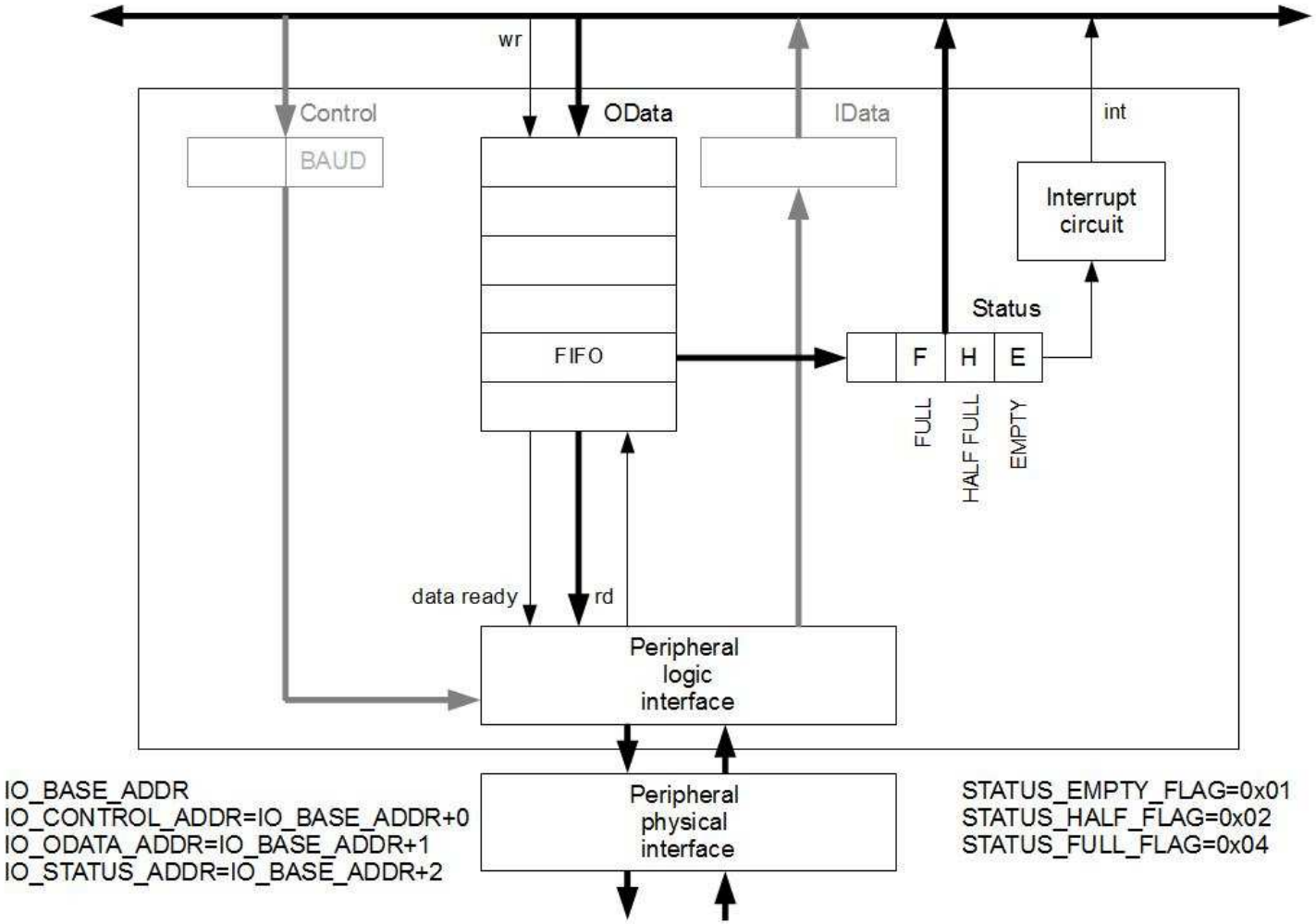
Polling for data send operation

```
#define IO_BASE_ADDR 0xEA0000000
#define IO_ODATA_ADDR IO_BASE_ADDR+1
#define IO_STATUS_ADDR IO_BASE_ADDR+2

#define STAT_BUSY_FLAG 0x01

volatile char* out_reg = IO_ODATA_ADDR;
volatile char* stat_reg = IO_STATUS_ADDR;

int Send(char* buf, int len)
{
int i=0;

    while(i<len)        //repeat until all characters from buffer send
    {
        while((*stat_reg&&STAT_BUSY_FLAG)!=0); //wait until device is ready for a new data

        *out_reg=buf[i]; //place next character in output register

        i++;            //incremant to next character in buffer
    }

    return 1;
}
```

Advanced control of IO device

- **A system can use interrupts to control peripherals**
- **Interrupt-driven I/O scheme employs interrupts to indicate to the processor that an I/O device needs attention.**
  - It is used by almost all systems
  - I/O interrupt is asynchronous with respect to the instruction execution
- **An interrupt mechanism eliminates the need for the processor to poll the device and instead allows the processor to focus on executing programs**
- **Device is programmed by a CPU to generate an interrupt in a certain events**
  - e,g, data is ready in receive buffer, transmit buffer is empty, an error occured
- **To eliminate too frequent interrupt events, IO communication device can use internal data buffering**

Buffering Data in Device



Interrupt Driven Communication Example

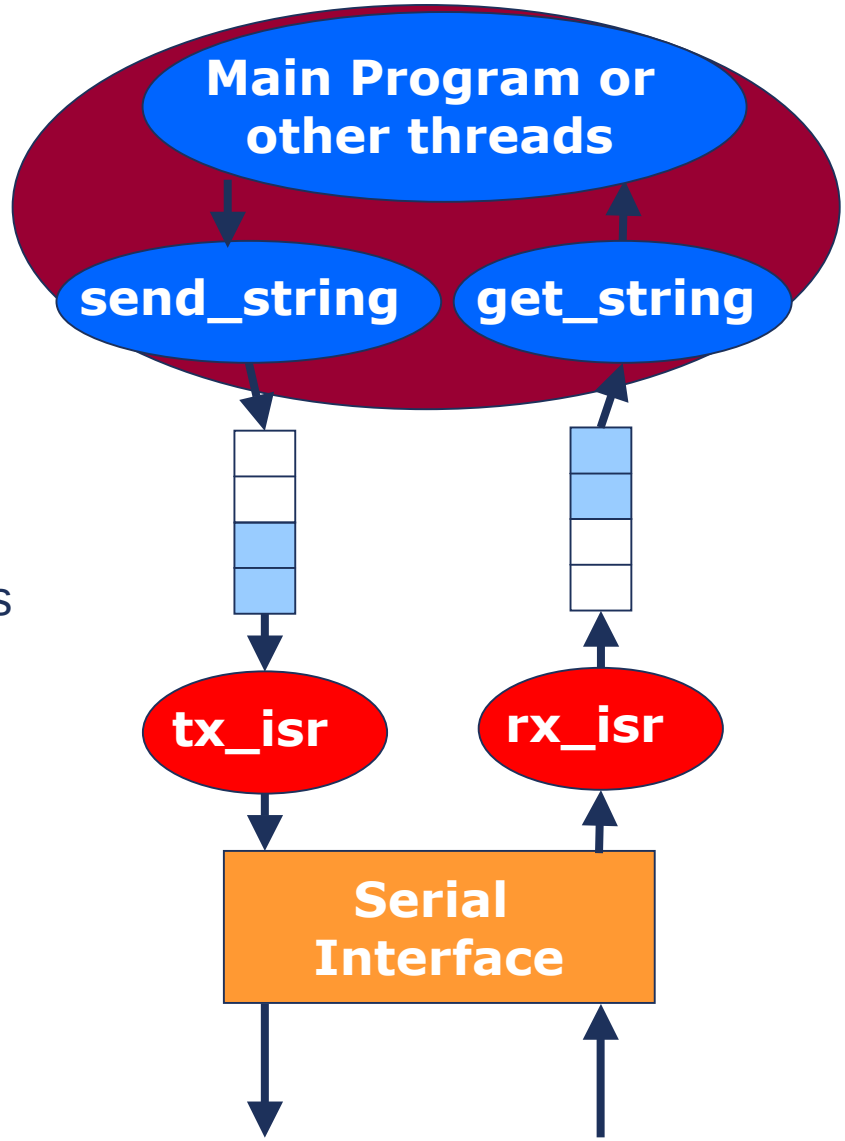
IO_interrupt_example.c

```
#define STAT_FULL_FLAG 0x04

volatile *char out_reg = IO_ODATA_ADDR;
volatile *char stat_reg = IO_STATUS_ADDR;

char *send_buffer;          //globally accessible buffer pointer
int length;                 //number of data to send
int i;                      //points to the next data in buffer

int Send(char* buf, int len)
{
    i=0;                    //initialize
    length=len;
    send_buffer=buf         //setup pointer to data
    start_io_interrupt();   //enable interrupt notification if fifo is empty
    return 1;               //return immediately
}

__irq void io_interrupt_handler (void) //interrupt service procedure;
{
    disable_all_interrupts();
    while(i<length)         //repeat until all characters from buffer send
    {
        if((*stat_reg && STAT_FULL_FLAG)!=0) break; //fill fifo with data until full

        *out_reg=buf[i];    //place character

        i++;                //incremant to next character in buffer
    }
    if(i==length) stop_io_interrupt();  //no further data so stop interrupting
    enable_all_interrupts();
}
```

SOFTWARE DATA BUFFERING FOR HANDLING ASYNCHRONOUS COMMUNICATION

Serial Communications and Interrupts

- **Want to provide multiple software modules 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 software buffering information between soft modules**
  - Solution: **circular queue** with head and tail pointers
  - One for tx, one for rx



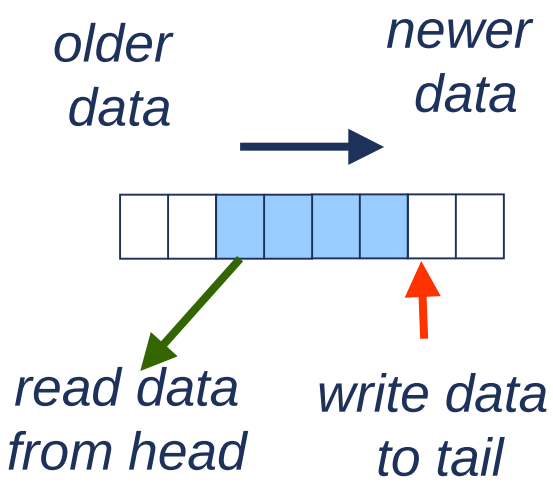
Enabling and Connecting Interrupts to ISRs

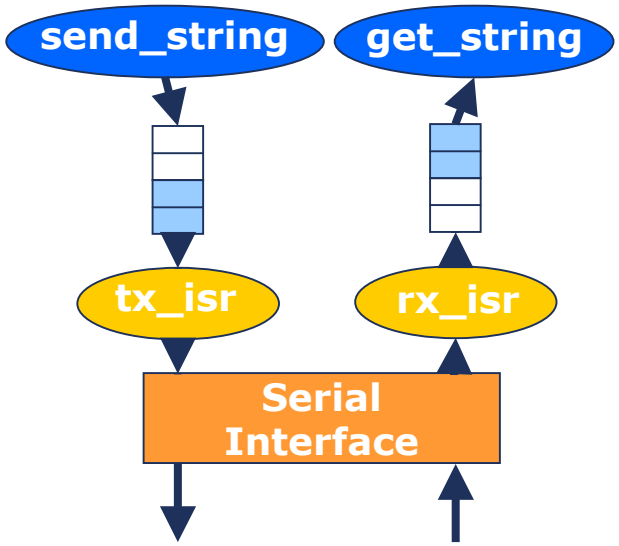
- ARM Cortex-M0+ provides one IRQ for all of a communication interface's events
- Within ISR (IRQ Handler), need to determine what triggered the interrupt, and then service it

```
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 (Q_SIZE) to make it easy to change**
- **One queue per direction**
  - tx ISR unloads tx_q
  - rx ISR loads rx_q
- **'Main' routine 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) {

  if (!Q_Full(q)) {    // What if queue is full?
    q->Data[q->Tail++] = d;
    q->Tail %= Q_SIZE; //Must wrap circular buffer
    q->Size++;
    return 1; // success
  } else
    return 0; // failure
}
unsigned char Q_Dequeue(Q_T * q) {

  unsigned char t=0;
  if (!Q_Empty(q)) {   // Must check to see if queue is empty
    t = q->Data[q->Head];
    q->Data[q->Head++] = 0; // to simplify debugging
    q->Head %= Q_SIZE; //Wrap circular buffer
    q->Size--;
  }
  return t;
}
```

Using the Queues

- **Sending data:**

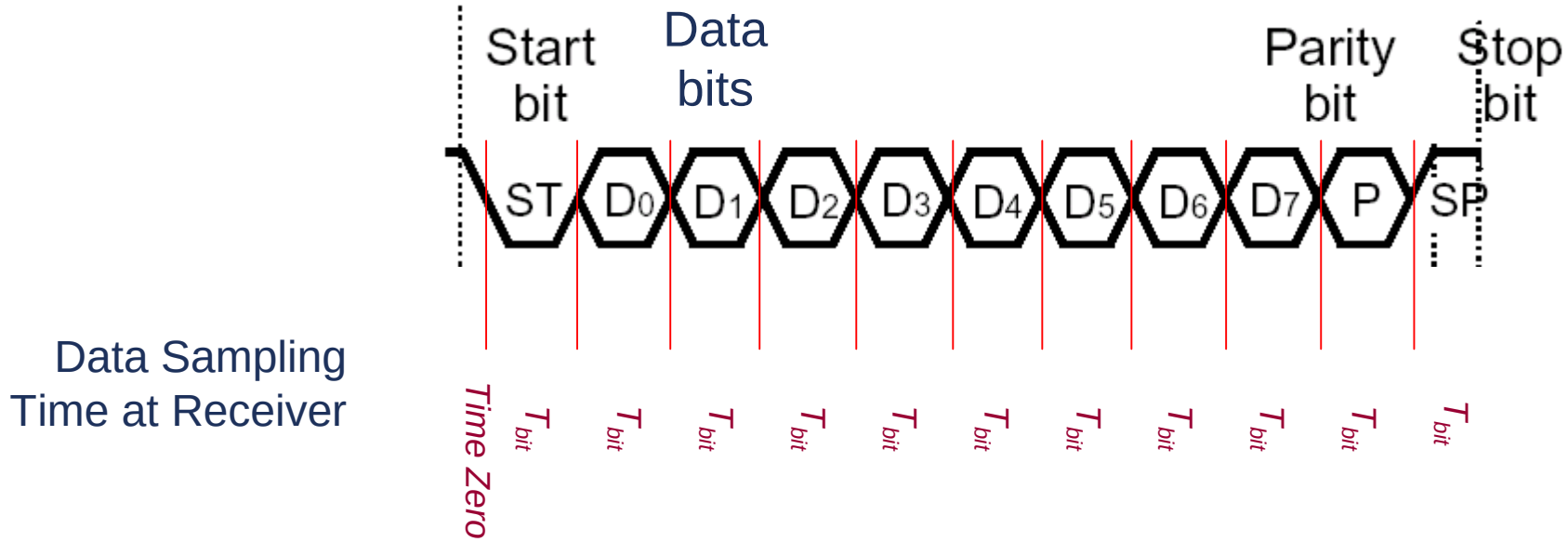
```
if (!Q_Full(…)) {
      Q_Enqueue(…, c)
}
```

- **Receiving data:**

```
if (!Q_Empty(…)) {
      c=Q_Dequeue(…)
}
```

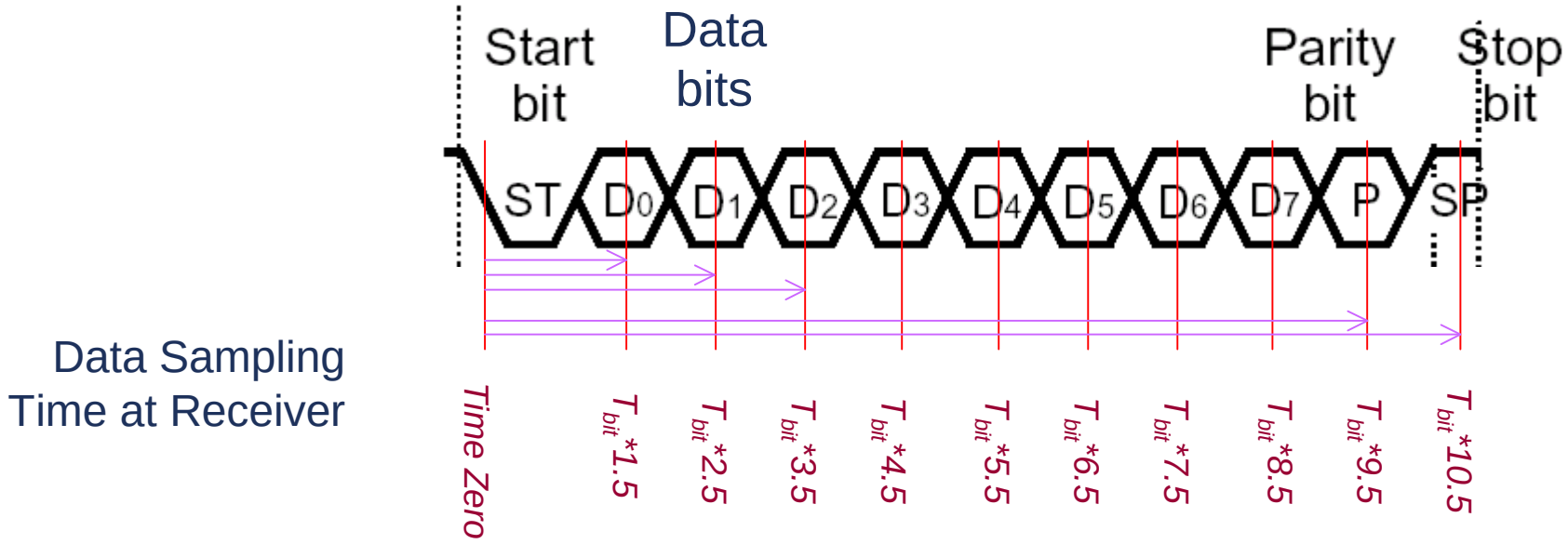
ASYNCHRONOUS SERIAL (UART) COMMUNICATIONS

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!

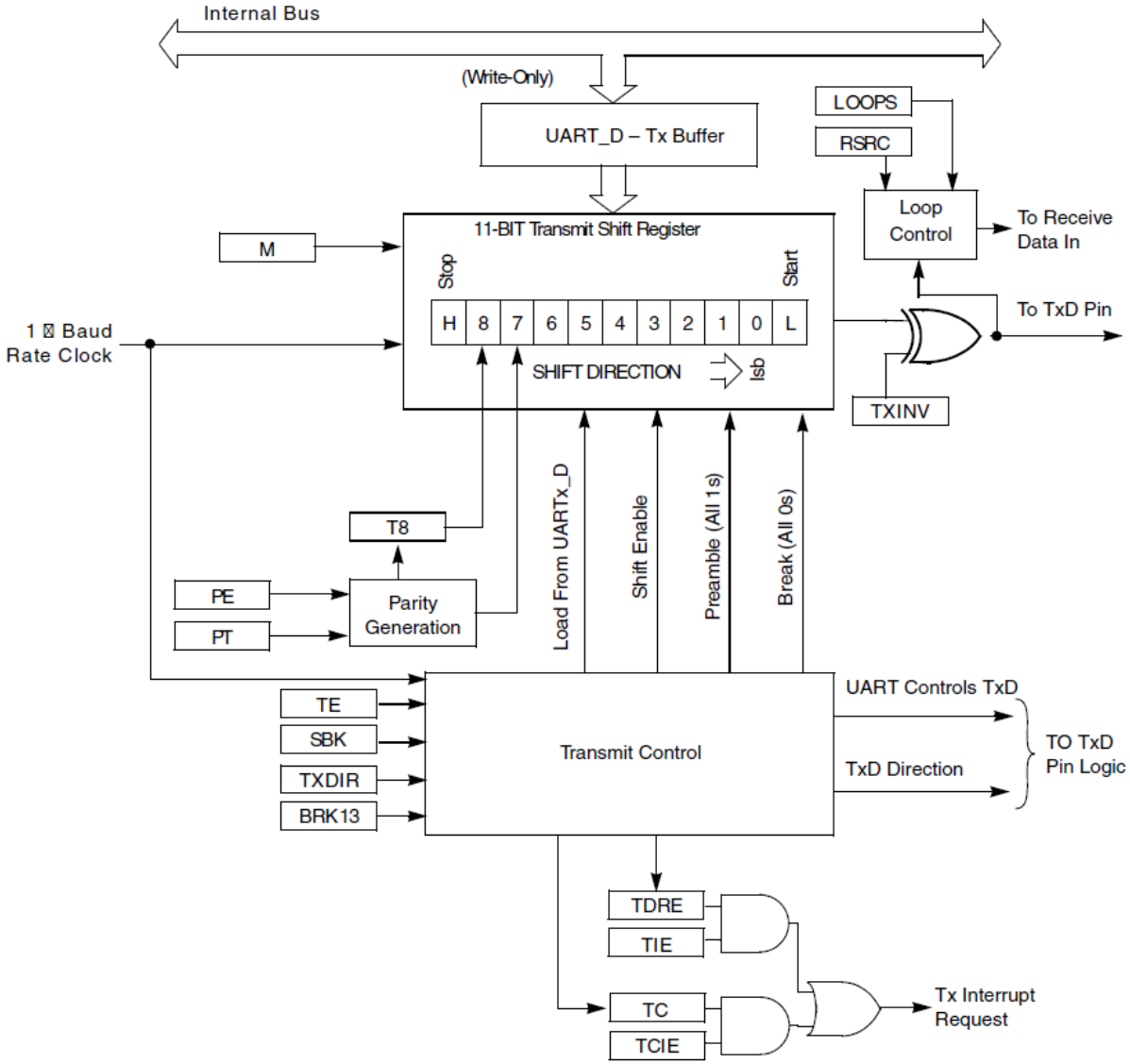
For this to work…

- **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, since the only timing reference is the start of the start bit

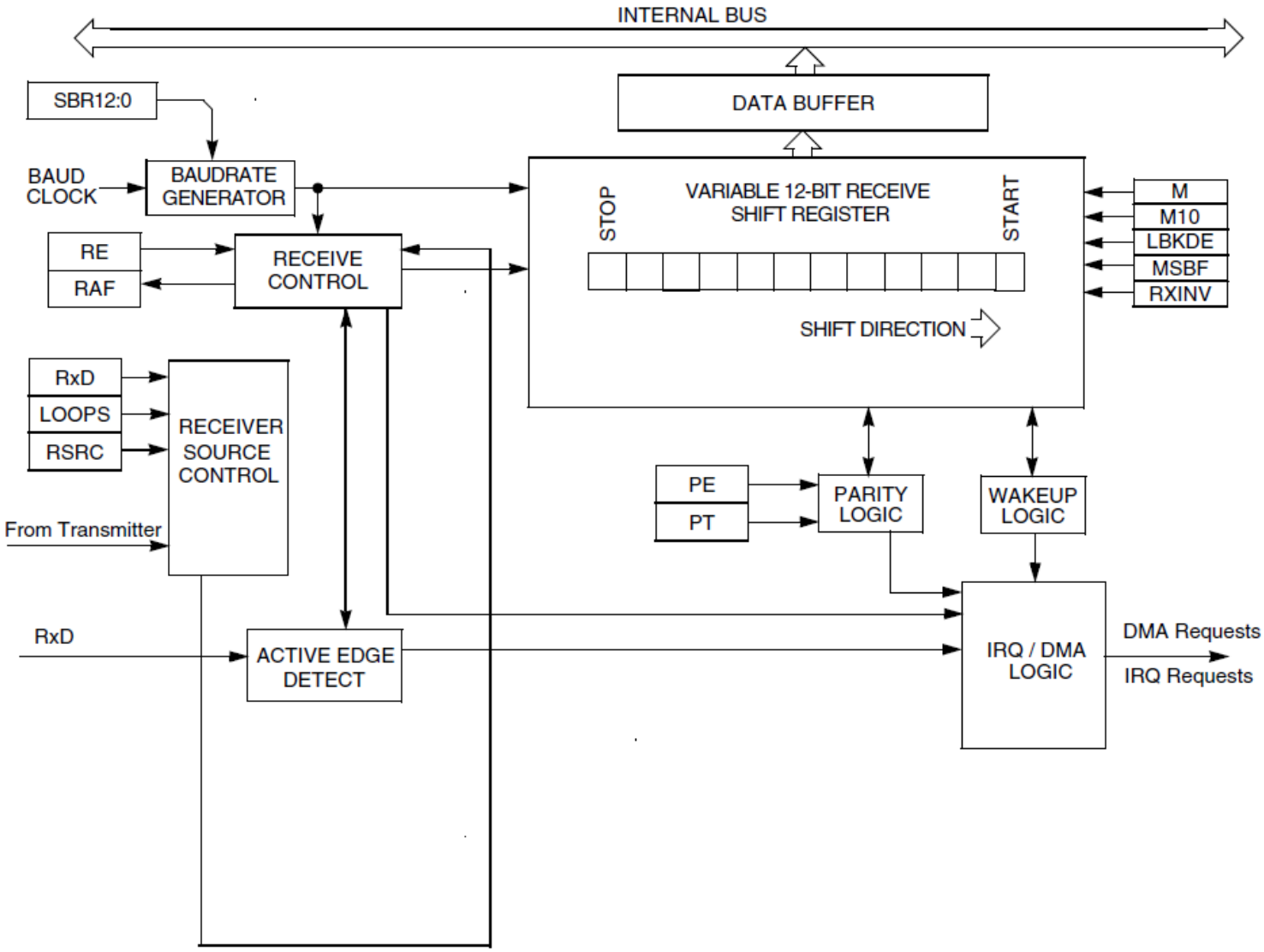
KL25 UARTs

- **UART: Universal (configurable) Asynchronous Receiver/Transmitter**
- **UART0**
  - Low Power
  - Can oversample from 4x to 32x
- **UART1, UART2**

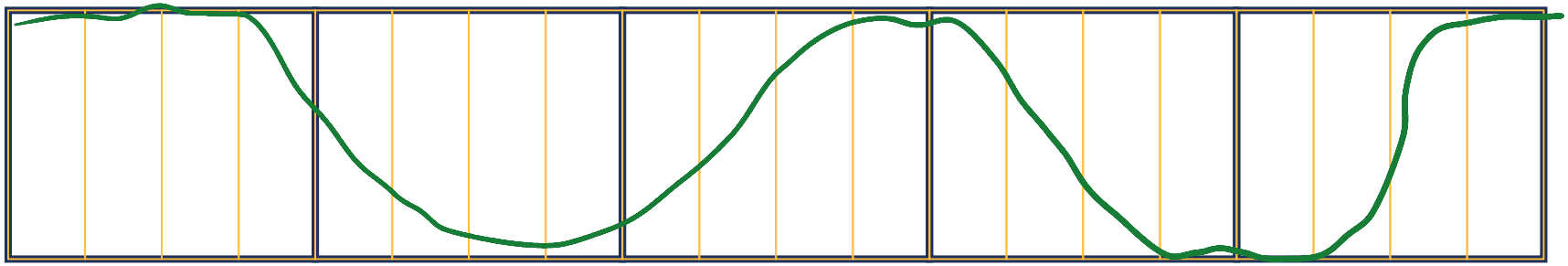
UART Transmitter



UART Receiver

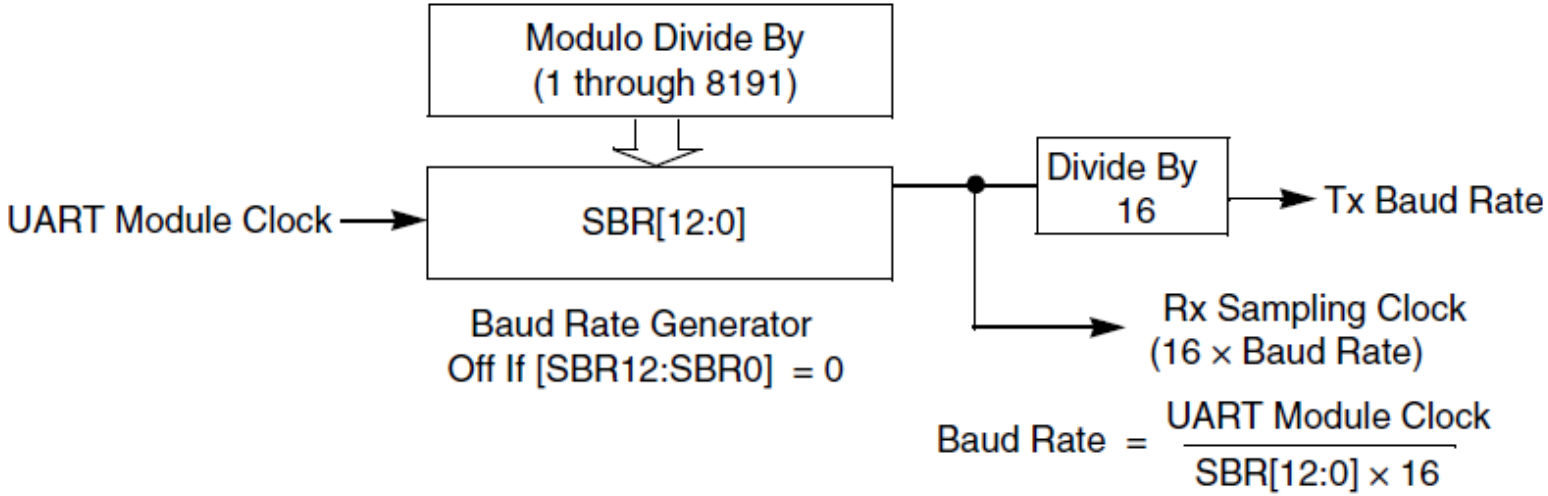


Input Data Oversampling



- **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)**

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;
}
```

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

```
char text[]="Hello world ...";
while (1) {
      for (c=0; c<=strlen(text); c++) {
            UART2_Transmit_Poll(text[c]);
      }
}
```

Example Receiver: Display Data on LCD

```
col = 0;
while (1) {
      c = UART2_Receive_Poll();
      lcd_putchar(c, col);
      col++;
      if (col>4) col = 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
            }
}
```

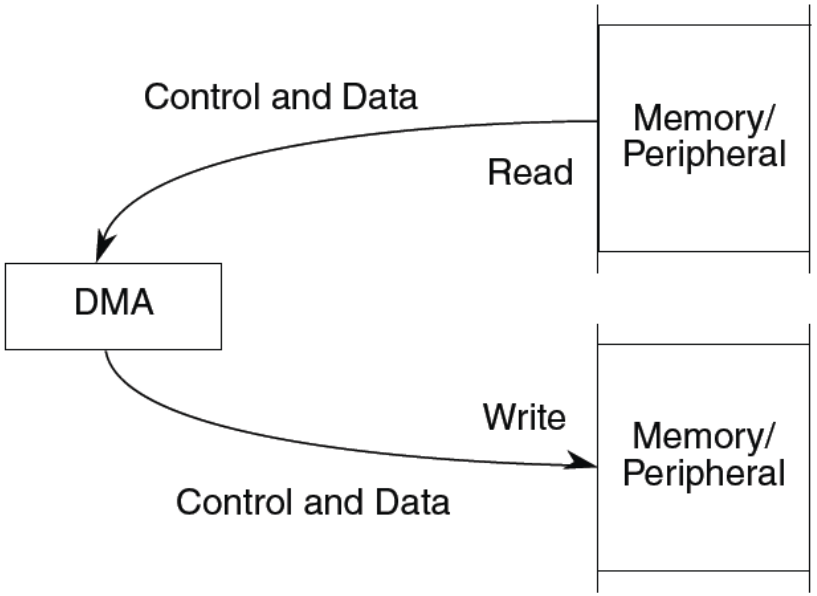
DIRECT MEMORY ACCESS OVERVIEW

Transferring data with DMA

- **For high-bandwidth devices, the transfers consist primarily of relatively large blocks of data**
  - overhead could still be intolerable, since it could consume a large fraction of the processor time
- **direct memory access (DMA) is a mechanism that provides the ability to transfer data directly to/from the memory without involving the processor**
  - Provided by the device controller
  - Provided by the central DMA device
- **DMA device must be a bus master**

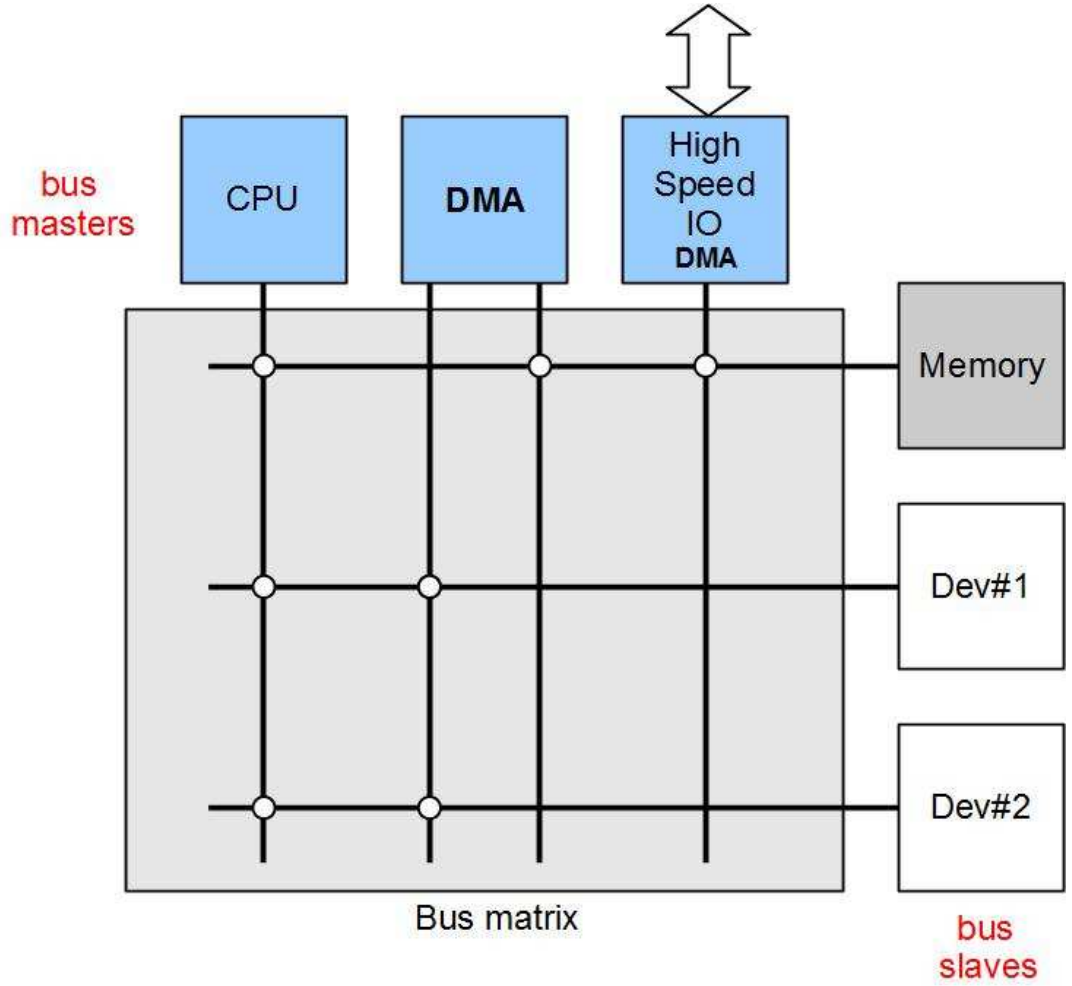
Basic Concepts

- **Special hardware to read data from a source and write it to a destination**
- **Various configurable options**
  - Number of data items to copy
  - Source and destination addresses can be fixed or change (e.g. increment, decrement)
  - Size of data item
  - When transfer starts
- **Operation**
  - Initialization: Configure controller
  - Transfer: Data is copied
  - Termination: Channel indicates transfer has completed



Example System with DMA engine

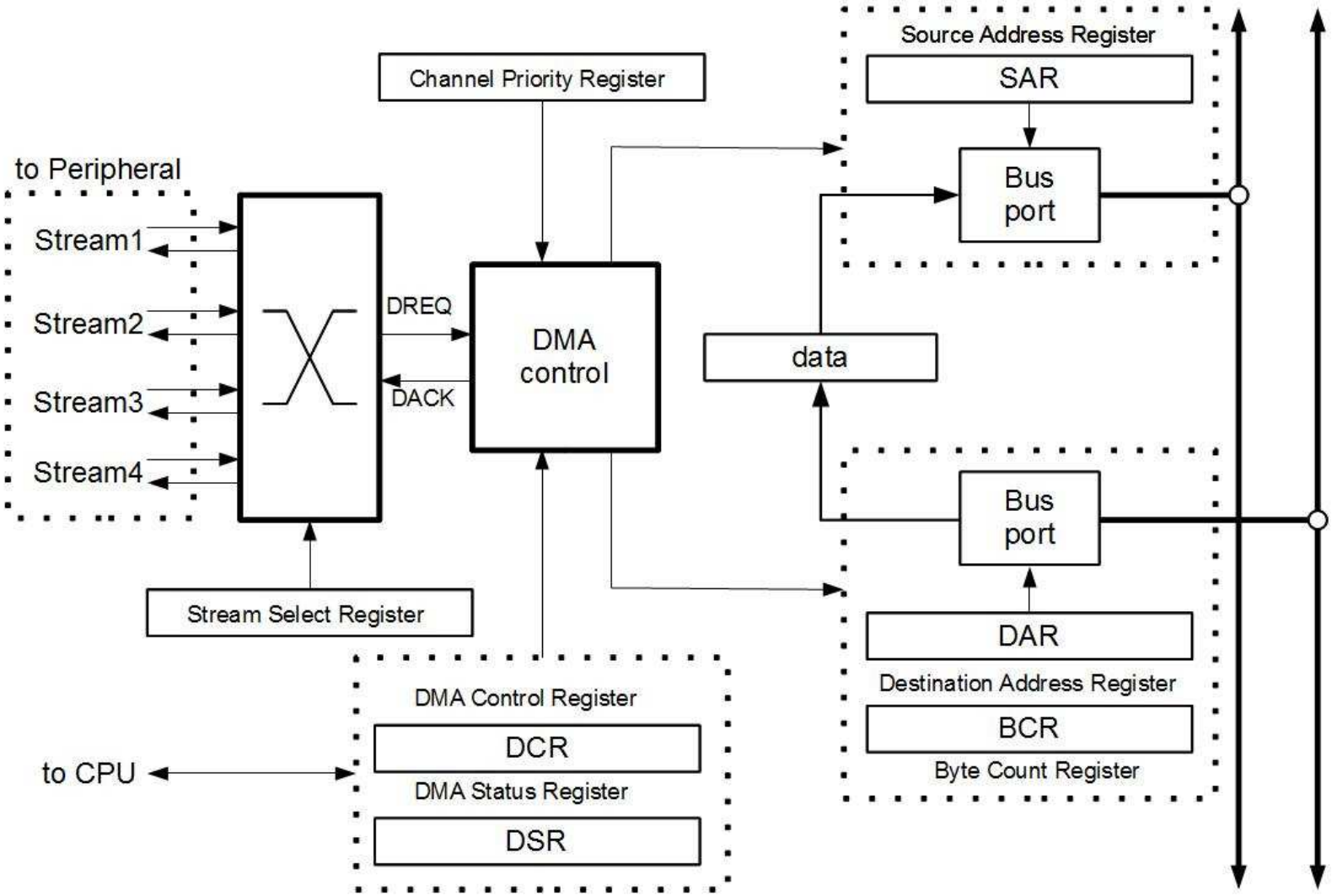
- Central or IO DMA engines are possible



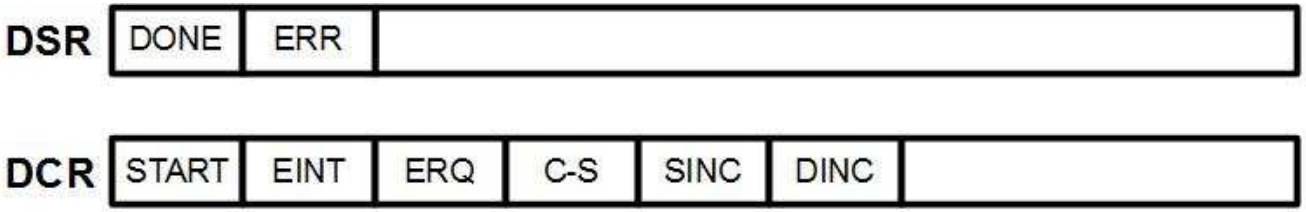
DMA basics

- **Direct memory access (DMA) is used in order to provide high-speed peripherals - memory and memory- memory data transfer.**
- **Each DMA transfer must have Transfer Control Descriptor (TCD) assigned**
- **TCD consist of**
  - **SAR** – Source Address Register, **DAR** - Destination Address Register, **DSR** - Status Register, **BCR** -Byte Count Register
- **DMA controller can provide multiple channels and it can service multiple sources / stereams**
  - Each channel has seperate **TDC**
  - Channels have different priority
  - Each peripheral is connected to DMA module using separate stream
  - Each channel request can be selected among possible stream requests

Central DMA structure example



DMA Status and Control flags



DONE - Transactions done. Set when all DMA controller transactions complete as determined by transfer count When BCR reaches zero, DONE is set when the final transfer completes successfully.

ERR - Error occured.

START - Start transfer.

EINT - Enable interrupt on completion of transfer.
Determines whether an interrupt is generated by completing a transfer or by the occurrence of an error Condition.

ERQ - Enable peripheral request.
Enables peripheral request to initiate transfer.

C-S - Cycle steal
0 DMA continuously makes read/write transfers until the BCR decrements to 0.
1 Forces a single read/write transfer per DMA request.

SINC - Source increment
Controls whether the source address increments after each successful transfer.

DINC - Destination increment

DMA procedure

- **The processor sets up the DMA by supplying:**
  - the identity of the device,
  - the operation to perform on the device,
  - the memory address that is the source or destination of the data to be transferred,
  - and the number of bytes to transfer
- **The DMA starts the operation on the device and arbitrates for the bus.**
  - DMA unit can complete an entire transfer, which may be thousands of bytes in length, without bothering the processor
- **Once the DMA transfer is complete, the controller interrupts the processor**

Transfer Requests

- **The DMA channel supports software-initiated or peripheral-initiated requests.**
  - A software request is issued by setting **DCR:START**
  - Peripheral request are initiated by asserting DMA Request (DREQ) signal when **DCR:ERQ** is set. Setting **DCR:ERQ** enables recognition of the peripheral DMA.
- **The hardware can be programmed to automatically clear DCR:ERQ, disabling the peripheral request, when BCR reaches zero.**

Cycle-steal and continuous modes

- **Cycle-steal mode (DCR:CS = 1)**
  - Only one complete transfer from source to destination occurs for each request.
- **Continuous mode (DCR:CS = 0)**
  - After a software-initiated or peripheral request, the DMA continuously transfers data until BCR reaches zero.
  - The DMA performs the specified number of transfers, then retires the channel.

Channel prioritization

- **Many DMA channels can be prioritized based on number, with channel 0 having the highest priority and the last channel having the lowest priority.**
- **Another scenario can assume the priority register to allow a programmers to set channel priorities according their need.**
  - Priorities between DMA stream requests are software-programmable (e.g. 4 levels consisting of very high, high, medium, low) or hardware in case of equality (request 0 has priority over request 1, etc.)

DMA termination

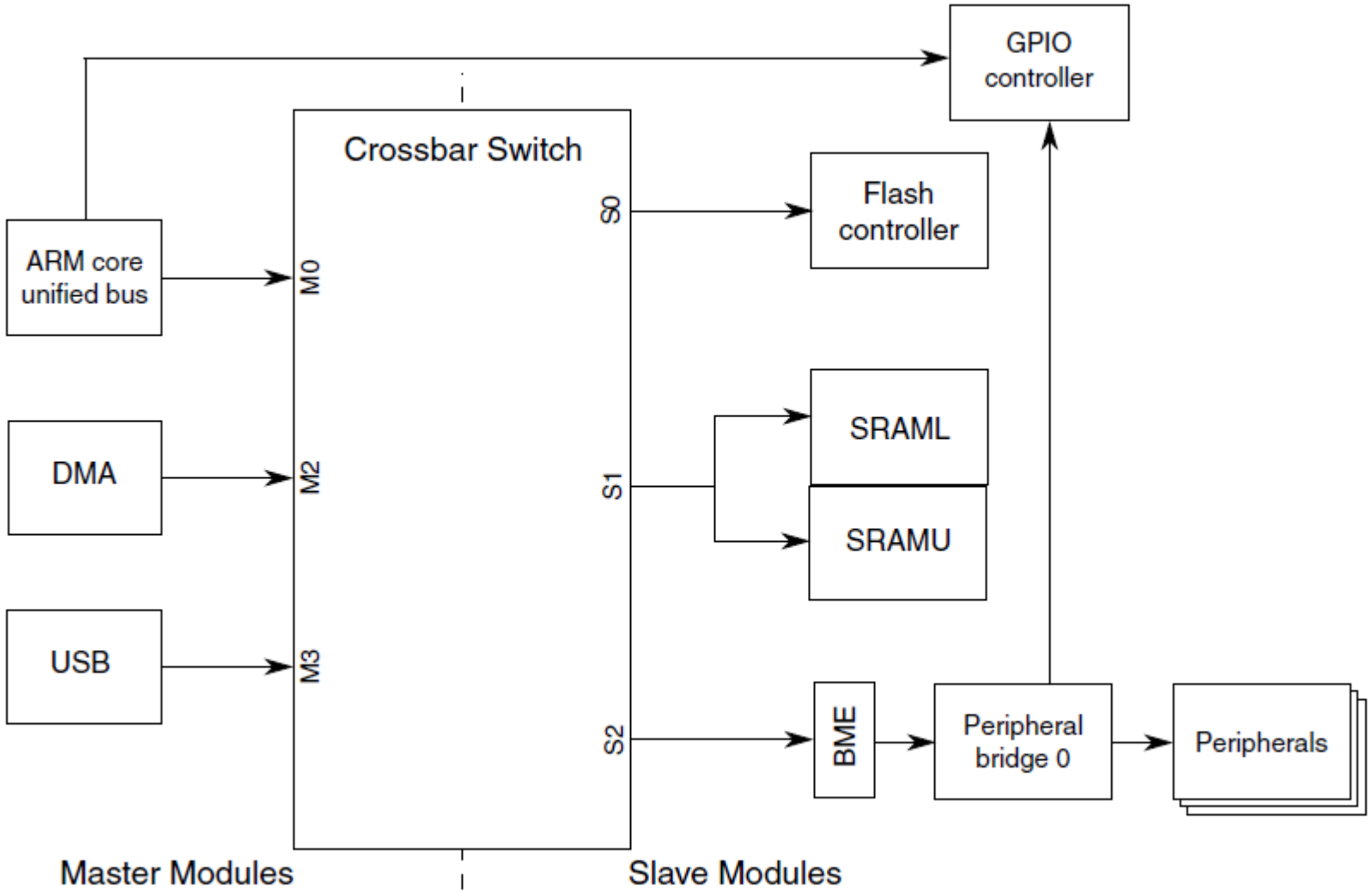
- **Interrupt**
  - If DCR:EINT is set, the DMA drives the appropriate interrupt request signal. The processor can read DSR to determine whether the transfer terminated successfully or with an error.

DMA MULTIPEXER AND MODULE IN KINETIS L

DMA Controller Features

- **4 independent channels**
  - Channel 0 has highest priority
- **8-, 16- or 32-bit transfers, data size can differ between source and destination**
- **Modulo addressable**
- **Can trigger with hardware signal or software**
- **Can run continuously or periodically (“cycle-stealing”)**
- **Hardware acknowledge/done signal**

Kinetis L family Crossbar Switch



DMA request multiplexer

- **The direct memory access multiplexer (DMAMUX) routes DMA sources, called slots, to any of the four DMA channels**
  - allows up to 63 DMA request signals

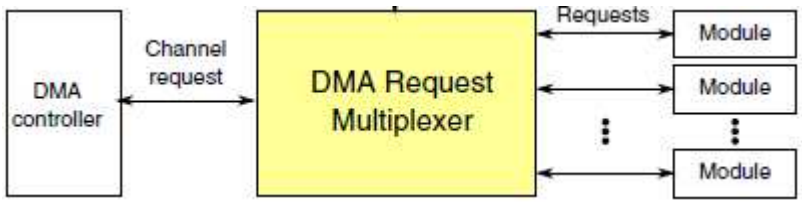
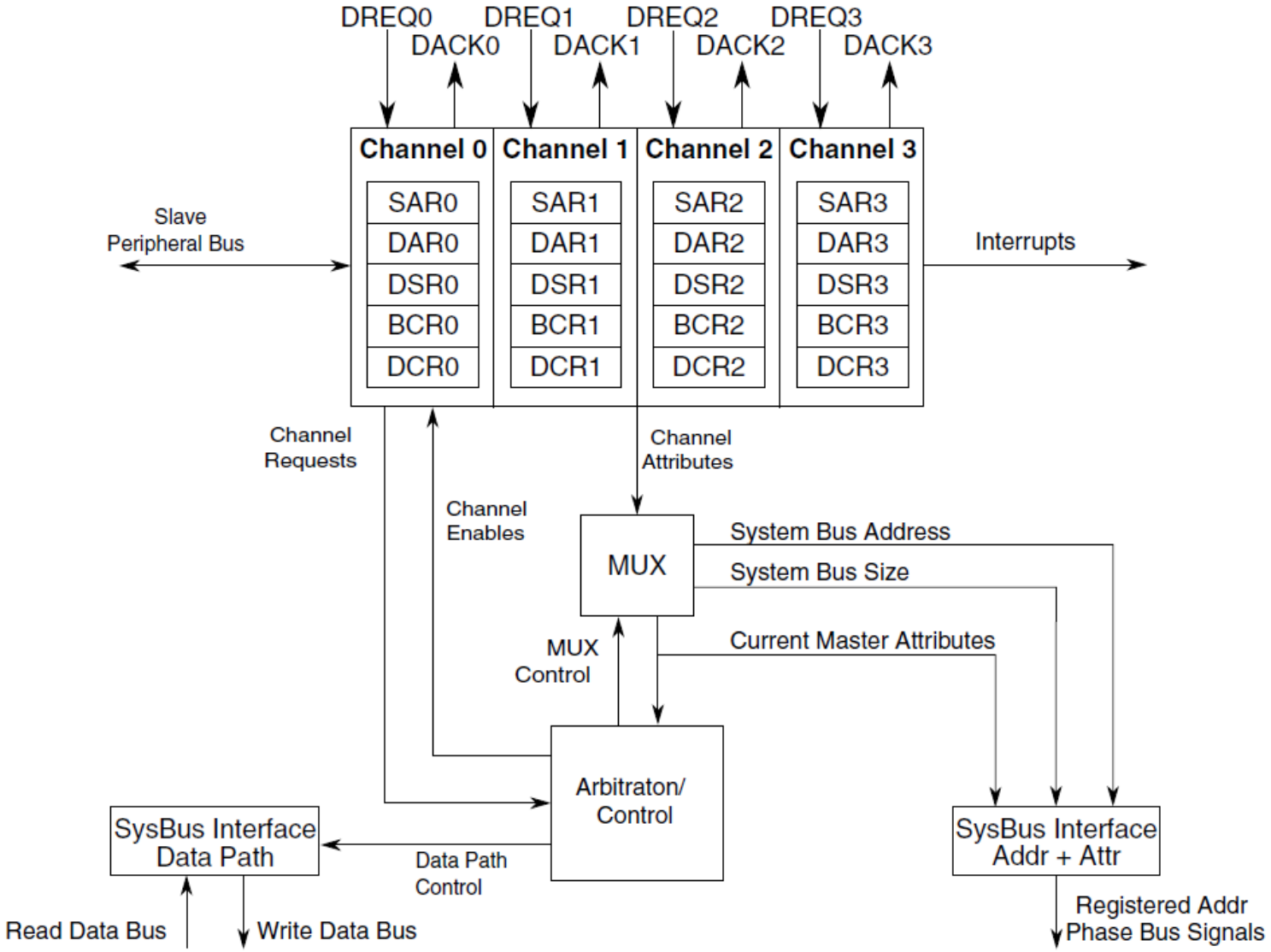


Table 3-20. DMA request sources - MUX 0

Source number	Source module	Source description	Async DMA capable
0	—	Channel disabled[1]	
1	Reserved	Not used	
2	UART0	Receive	Yes
3	UART0	Transmit	Yes
4	UART1	Receive	
5	UART1	Transmit	
6	UART2	Receive	
7	UART2	Transmit	

Table continues ...

Kinteis L DMA Controller

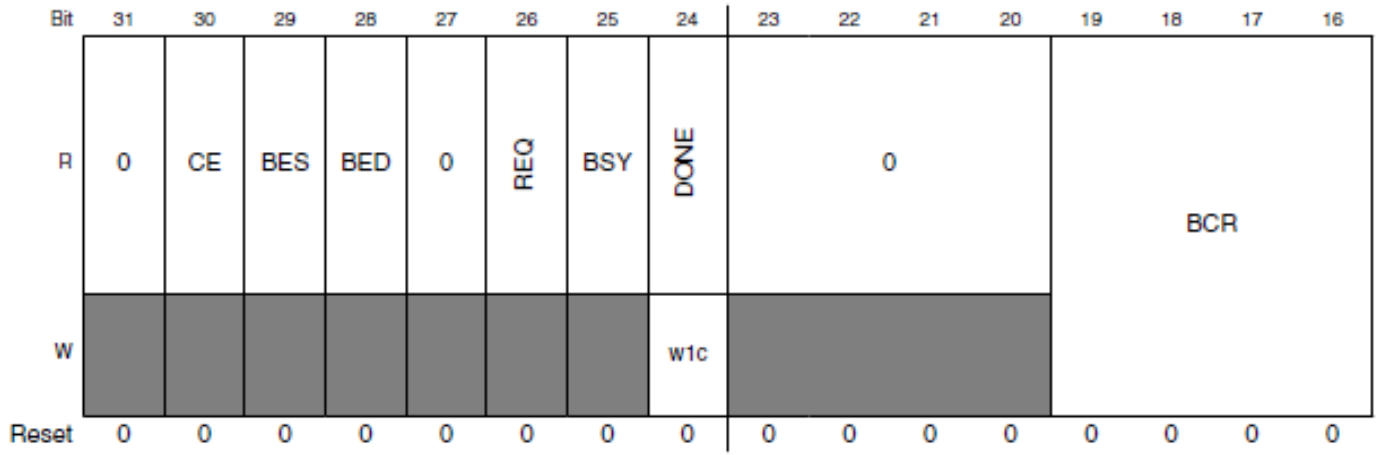


Registers

- **DMA_SARn**
  - Source address register,
  - Valid values 0 to 0x000f ffff

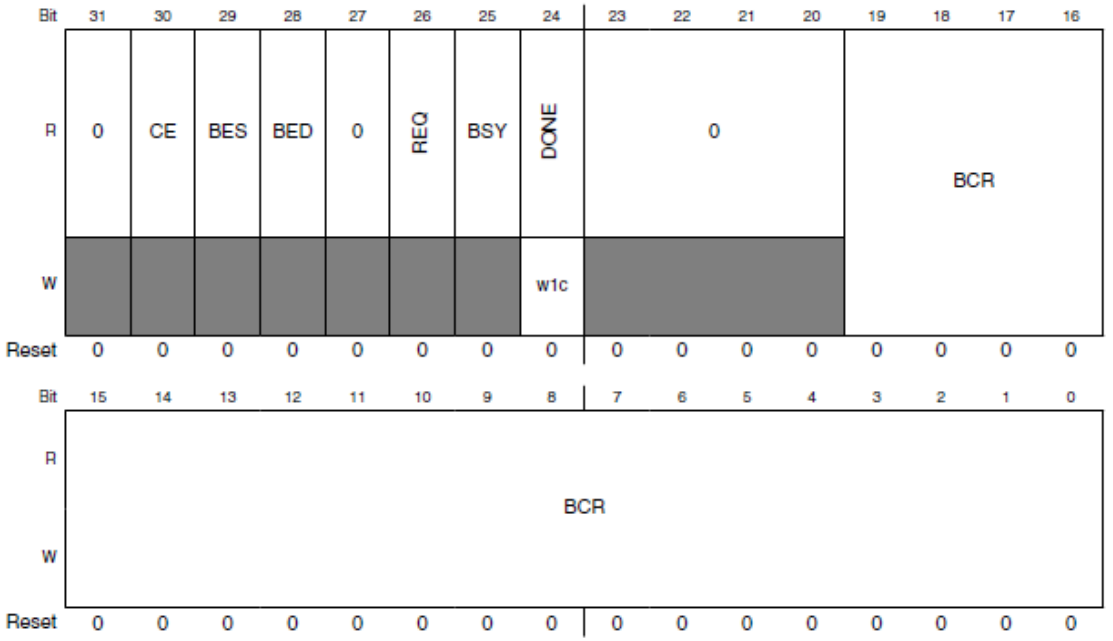
- **DMA_DARn**
  - Destination address register
  - Valid values 0 to 0x000f ffff

Status Register/Byte Count Register DMA_DSR_BCRn



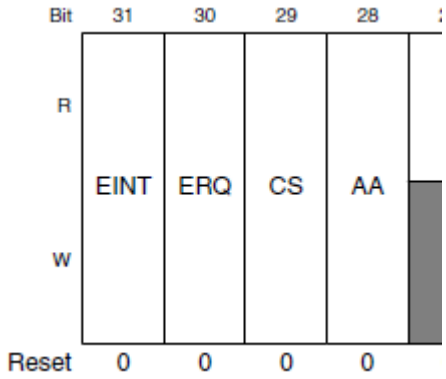
- **Status flags: 1 indicates error**
  - CE: Configuration error
  - BES: Bus error on source
  - BED: Bus error on destination
  - REQ: A transfer request is pending (more transfers to perform)
  - BSY: DMA channel is busy
  - DONE: Channel transfers have completed or an error occurred. Clear this bit in an ISR.

Byte Count Register



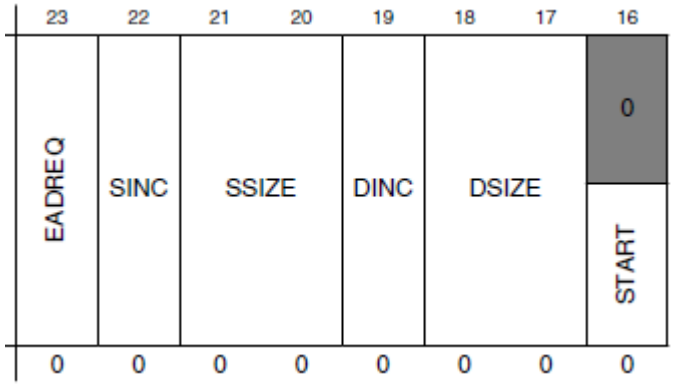
- **BCR: Bytes remaining to transfer**
- **Decremented by 1, 2 or 4 after completing write (determined by destination data size)**

DMA Control Register (DMA_DCRn)



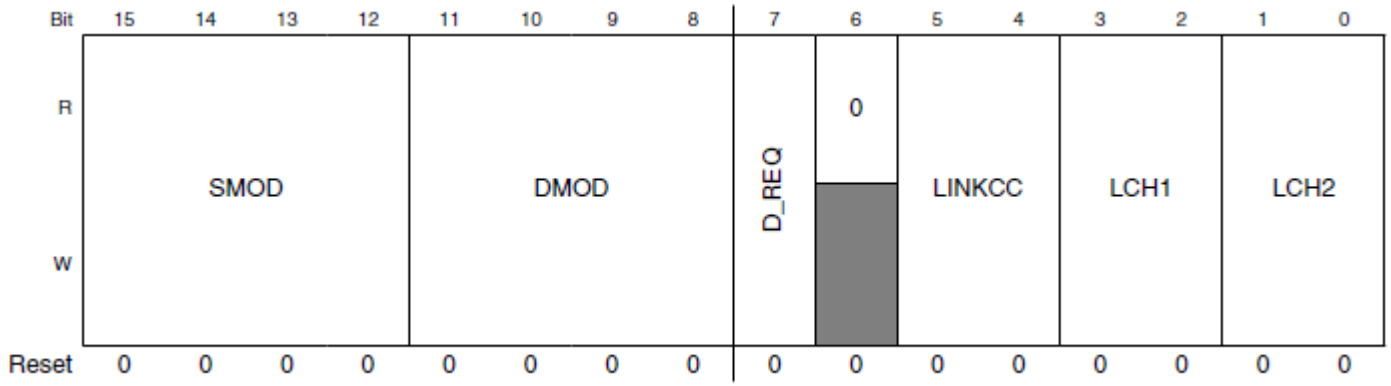
- **EINT: Enable interrupt on transfer completion**
- **ERQ: Enable peripheral request to start transfer**
- **CS: Cycle steal**
  - 0: Greedy - DMA makes continuous transfers until BCR == 0
  - 1: DMA shares bus, performs only one transfer per request
- **AA: Autoalign**

DMA Control Register (DMA_DCRn)



- **EADRQ – Enable asynchronous DMA requests when MCU is in Stop mode**
- **SINC/DINC – Increment SAR/DAR (by 1,2 or 4 based on SSIZE/DSIZE)**
- **SSIZE/DSIZE – Source/Destination data size.**
  - Don't need to match – controller will perform extra reads or writes as needed (e.g. read one word, write two bytes).
  - 00: longword (32 bits)
  - 01: byte (8 bits)
  - 10: word (16 bits)
- **START – Write 1 to start transfer**

DMA Control Register (DMA_DCRn)



- **SMOD, DMOD – Source/Destination address modulo**
  - When non-zero, supports circular data buffer – address wraps around after 2^{n+3} bytes (16 bytes to 64 kilobytes)
  - When zero, circular buffer is disabled
- **D_REQ: If 1, then when BCR reaches zero ERQ bit will be cleared**
- **LINKCC: Enables this channel to trigger another channel**
  - 00: Disabled
  - 01: Two stages:
    - Link to channel LCH1 after each cycle-steal transfer
    - Link to channel LCH2 after BCR reaches 0
  - 10: Link to channel LCH1 after each cycle-steal transfer
  - 11: Link to channel LCH1 after BCR reaches 0
- **LCH1, LCH2: Values 00 to 11 specify linked DMA channel (0 to 3)**

Basic Use of DMA

- **Enable clock to DMA module (in SIM register SCGC7)**
- **Initialize control registers**
- **Load SARn with source address**
- **Load DARn with destination address**
- **Load BCRn with number of bytes to transfer**
- **Clear DSRn[DONE]**
- **Start transfer by setting DCRn[START]**
- **Wait for end of transfer**
  - Interrupt generated if DCRn[EINT] is set (DMAn_IRQHandler)
  - Poll DSRn[DONE]

Demonstration: Memory Copy

- **Software-triggered**
- **void Copy_Longwords(uint32_t * source, uint32_t * dest, uint32_t count)**

- **Could use as a fast version of memcpy function, but need to handle all cases**
  - Alignment of source and destination
  - Data size
  - Detecting overlapping buffers

Memory to memory DMA Initialization

```
#include <stdint.h>
#include <MKL25Z4.h>

void Init_DMA_To_Copy(void) {
  SIM->SCGC7 |= SIM_SCGC7_DMA_MASK;
  DMA0->DMA[0].DCR =   DMA_DCR_SINC_MASK |
                       DMA_DCR_SSIZE(0) |
                       DMA_DCR_DINC_MASK |
                       DMA_DCR_DSIZE(0);
            // Size: 0 = longword, 1 = byte, 2 = word

}
```

Memory to memory copy using DMA

```
void Copy_Longwords( uint32_t *source,
                     uint32_t * dest,
                     uint32_t count) {
  // initialize source and destination pointers
  DMA0->DMA[0].SAR = DMA_SAR_SAR((uint32_t) source);
  DMA0->DMA[0].DAR = DMA_DAR_DAR((uint32_t) dest);
  // byte count
  DMA0->DMA[0].DSR_BCR = DMA_DSR_BCR_BCR(count*4);
  // verify done flag is cleared
  DMA0->DMA[0].DSR_BCR &= ~DMA_DSR_BCR_DONE_MASK;

  // start transfer
  DMA0->DMA[0].DCR |= DMA_DCR_START_MASK;
  // wait until it is done
  while (!(DMA0->DMA[0].DSR_BCR & DMA_DSR_BCR_DONE_MASK))
      ;
}
```