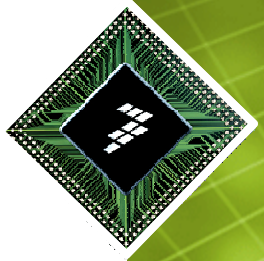
UART

Universal Asynchronous Receiver/Transmitter



Agenda

1. Overview
2. Diagram Connections
3. Software Configuration
4. UART Hands-On

Agenda

1. Overview
2. Diagram Connections
3. Software Configuration
4. UART Hands-On

- **Overview:**
  - Communication interfaces enables communications between systems of different architectures. The Universal asynchronous receiver/transmitter (UART) peripheral is one of the many interfaces available in the Kinetis KL25. The UART will be used in this lab to communicate with a terminal (TeraTerm).
- **Learning Objectives:**
  - This module will help you learn how to use the Kinetis KL25 UART with a terminal on a PC using CDC OpenSDA.
- **Success Criteria:**
  - When you have completed this module, you will be able to have the KL25 communicate with a PC through the OpenSDA CDC Serial Port.

Features

- Full-duplex
- Double-buffered transmitter and receiver with separate enables
- Programmable baud rates (13-bit modulo divider)
- Transmit and receive baud rate can operate asynchronous to the bus clock
- Interrupt, DMA(UART0) or polled operation
- Hardware parity generation and checking
- Programmable 8-bit, 9-bit or 10-bit character length
- Programmable 1-bit or 2-bit stop bits
- Receiver wakeup by idle-line, address-mark or address match
- Optional 13-bit break character generation / 11-bit break character detection
- Selectable transmitter output and receiver input polarity

Agenda

1. Overview
2. Diagram Connections
3. Software Configuration
4. UART Hands-On

Diagram Connections – Clock Source

- The UART0 module has a selectable clock as shown in the following figure. UART1 and UART2 modules operate from the bus clock.

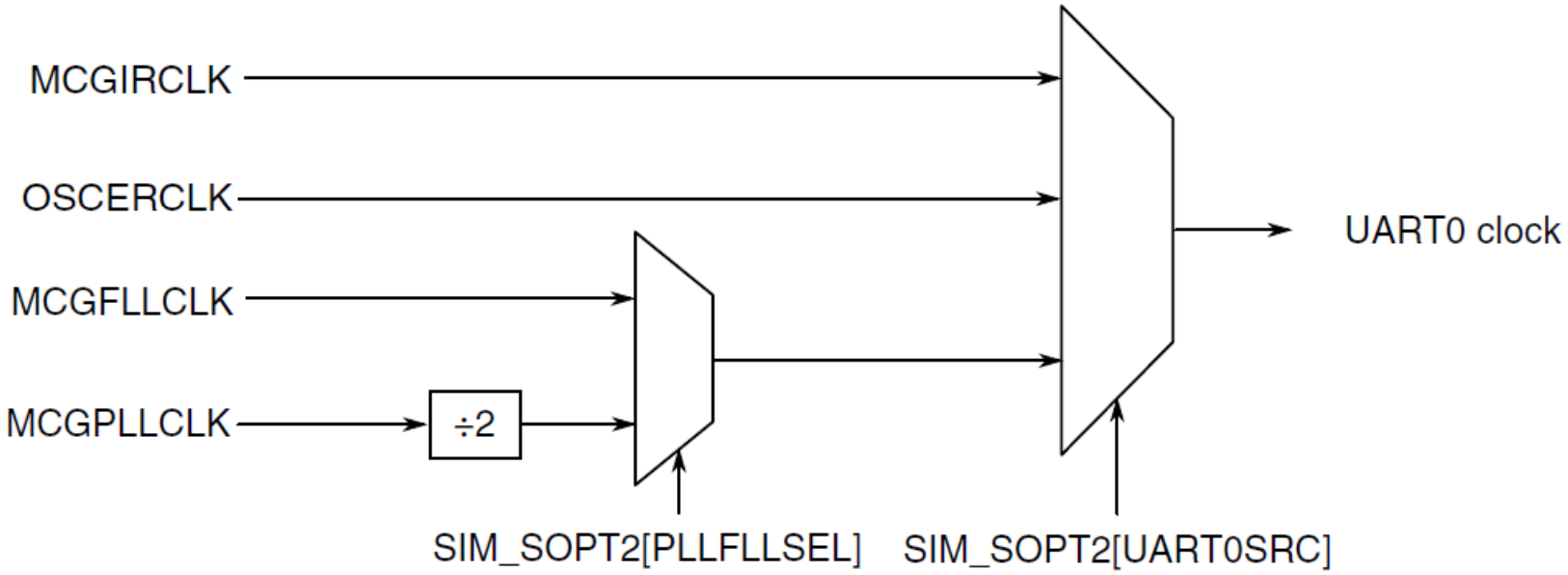


Diagram Connections – TX block diagram

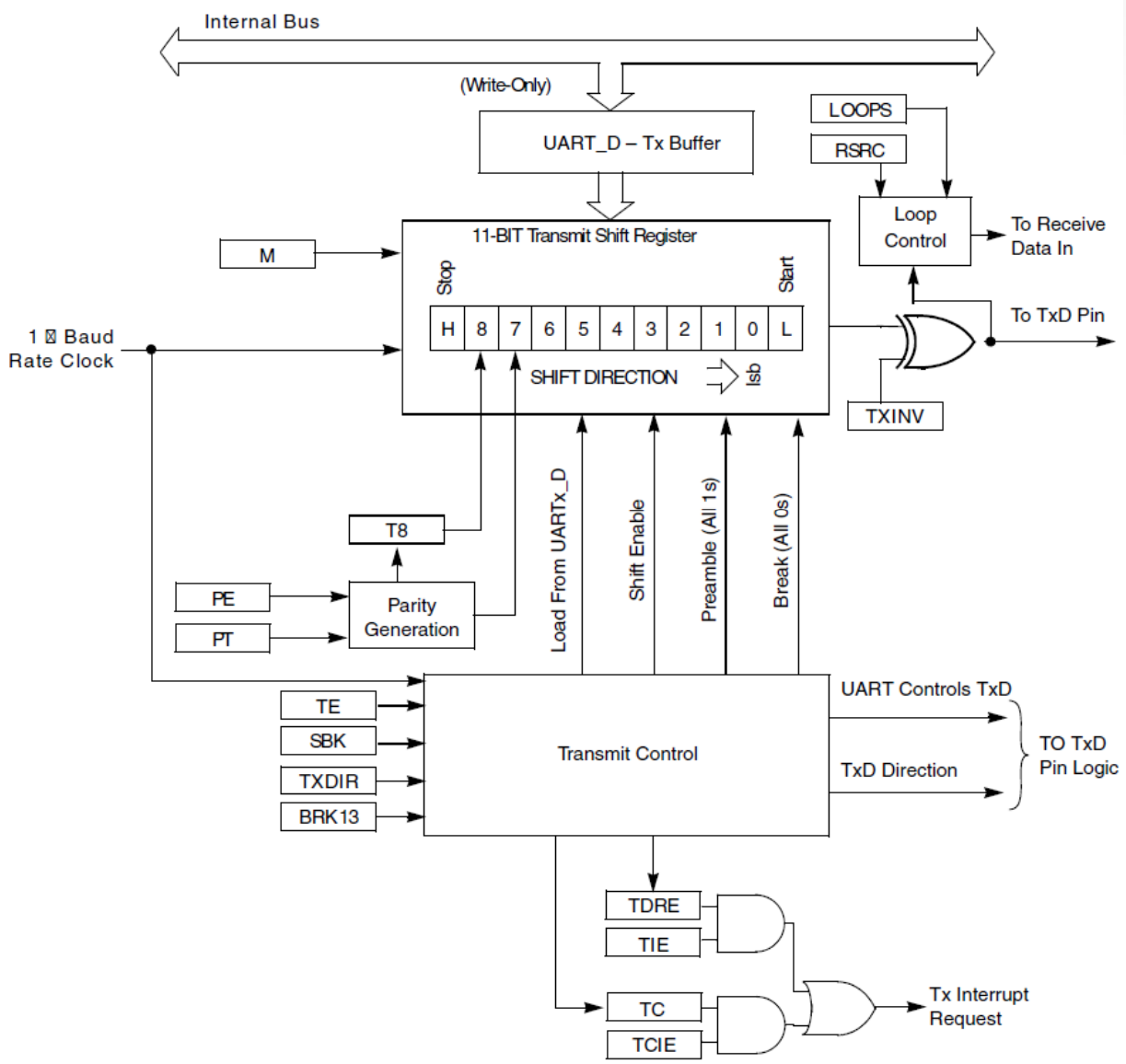
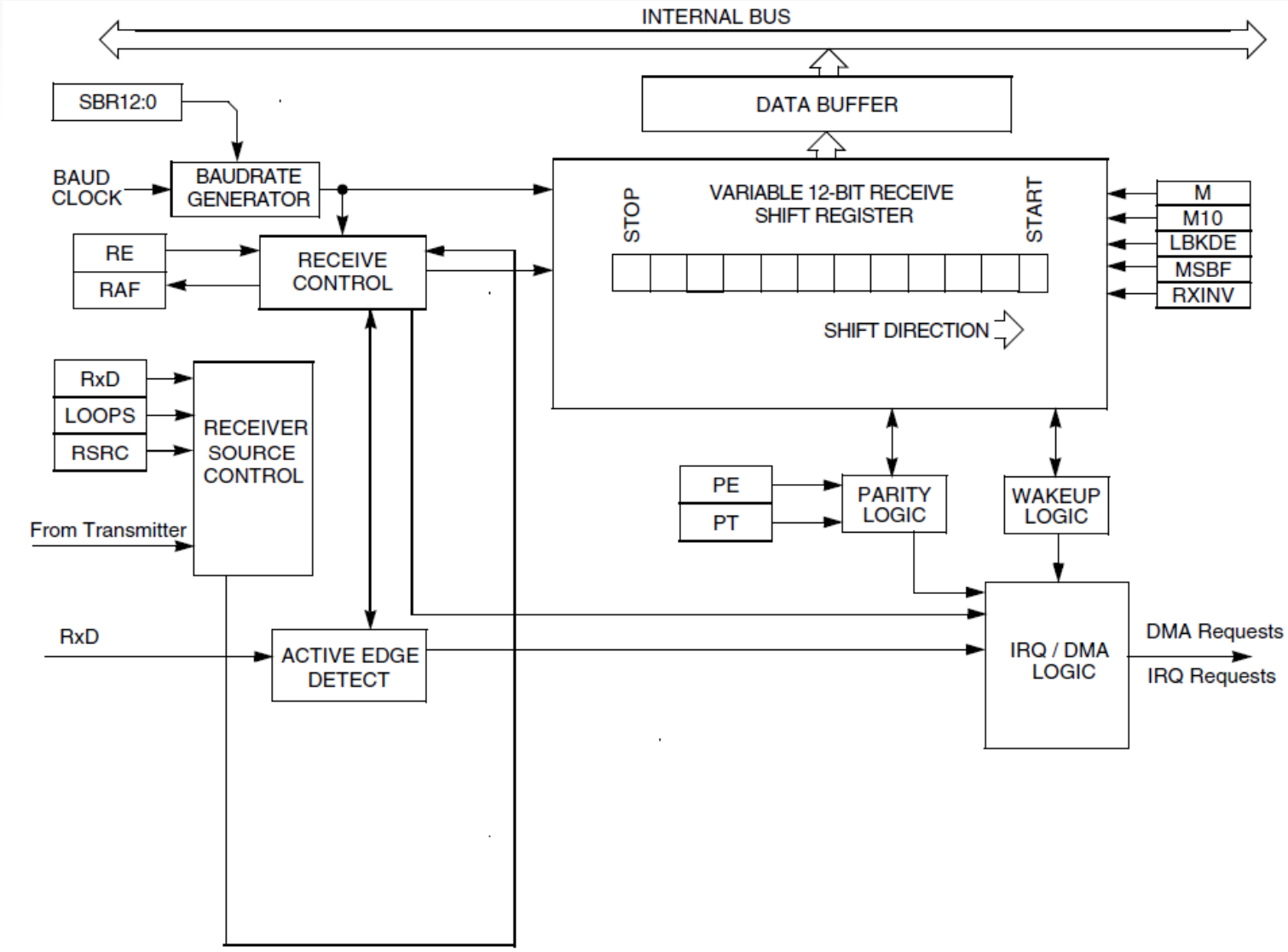


Diagram Connections – RX block diagram



Agenda

1. Overview
2. Diagram Connections
3. Software Configuration
4. UART Hands-On

Baud Rate Generation – UART0

- The Kinetis KL25 UART0 module uses a 13-bit modules counter and 5-bit Oversampling Ratio to generate the baudrate.
- Baud Rates are calculated by the following equation:

$$Baud\ Rate = \frac{UART\ Module\ Clock}{((OSR[4:0]+1)\times SBR[12:0])}$$

- SBR[12:0] is the Baud Rate Modulo Divisor
- OSR[4:0] is the Over Sampling Ratio

Baud Rate Generation – UART1/2

- The Kinetis KL25 UART1/2 module uses a 13-bit modules counter to generate the baudrate.
- Baud Rates are calculated by the following equation:

$$Baud\ Rate = \frac{UART\ Module\ Clock}{(SBR[12:0] \times 16)}$$

- SBR[12:0] is the Baud Rate Modulo Divisor

Baud Rate Generation

- The UART0 baud rate is configured using three registers, UART0_BDH, UART0_BDL and UART0_C5.
- BDH and BDL are used to configure the 13-bit SBR field, and C5 is used to configure the Over Sampling Rate.

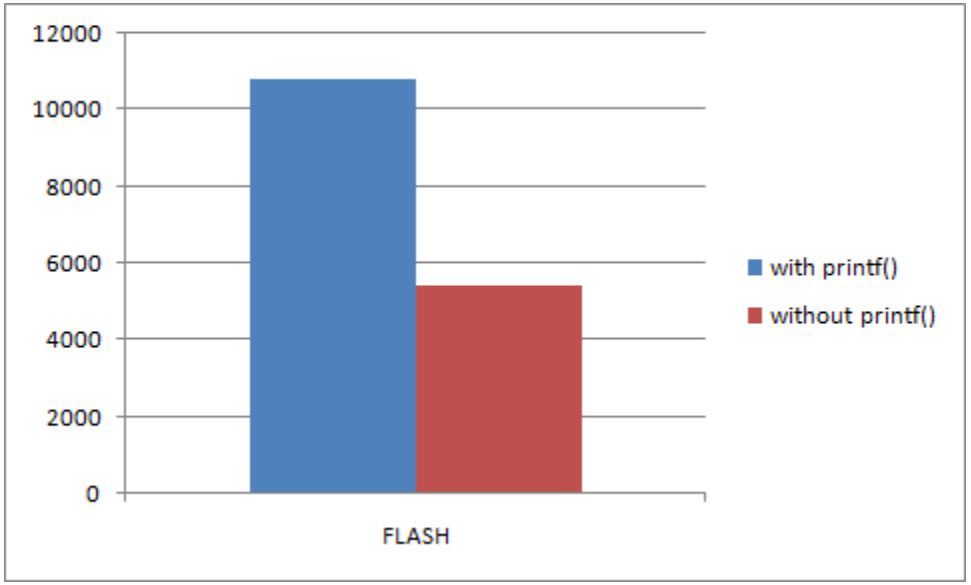
- The UART1/2 baud rate is configured using two registers, UART0_BDH and UART0_BDL.

Polling, Interrupt, or DMA Configuration

- The UART module can be configured to handle data flow by polling status flag, generating interrupts, or using the DMA(Only UART0).
- **Polling** is the most CPU intensive, but it might make the most sense when handling small messages.
- The UART status **interrupt** can be used to decrease CPU loading. Status Interrupts conditions are:
  - TDRE - Transmit Data Register Empty Flag
  - TC - Transmission Complete Flag
  - RDRF - Receive Data Register Full Flag
  - IDLE - Idle Line Flag
  - RXEDGIF -
  - LBKDIF - LIN Break Detect Interrupt Flag
  - OR - Receiver Overrun Flag
  - NF - Noise Flag
  - FE - Framing Error Flag
  - PF - Parity Error Flag
- The DMA can be used to automatically move receive and/or transmit data to reduce CPU loading even more.

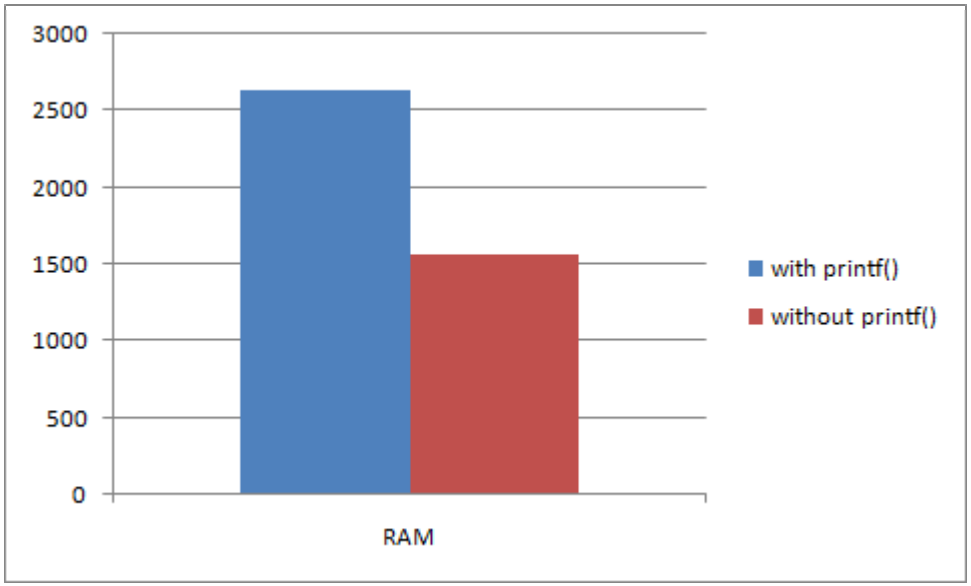
Please Do *not* use printf()

- **Code Size:**
  - Using printf() adds greatly to the code size of the application. I have seen cases where this is in the range of **10-20 KByte** of code. The problem comes from the fact that printf(), as defined by the ANSI library standard, needs to support all the different format string. Including formatting octal numbers, floating point, etc.



Please Do *not* use printf()

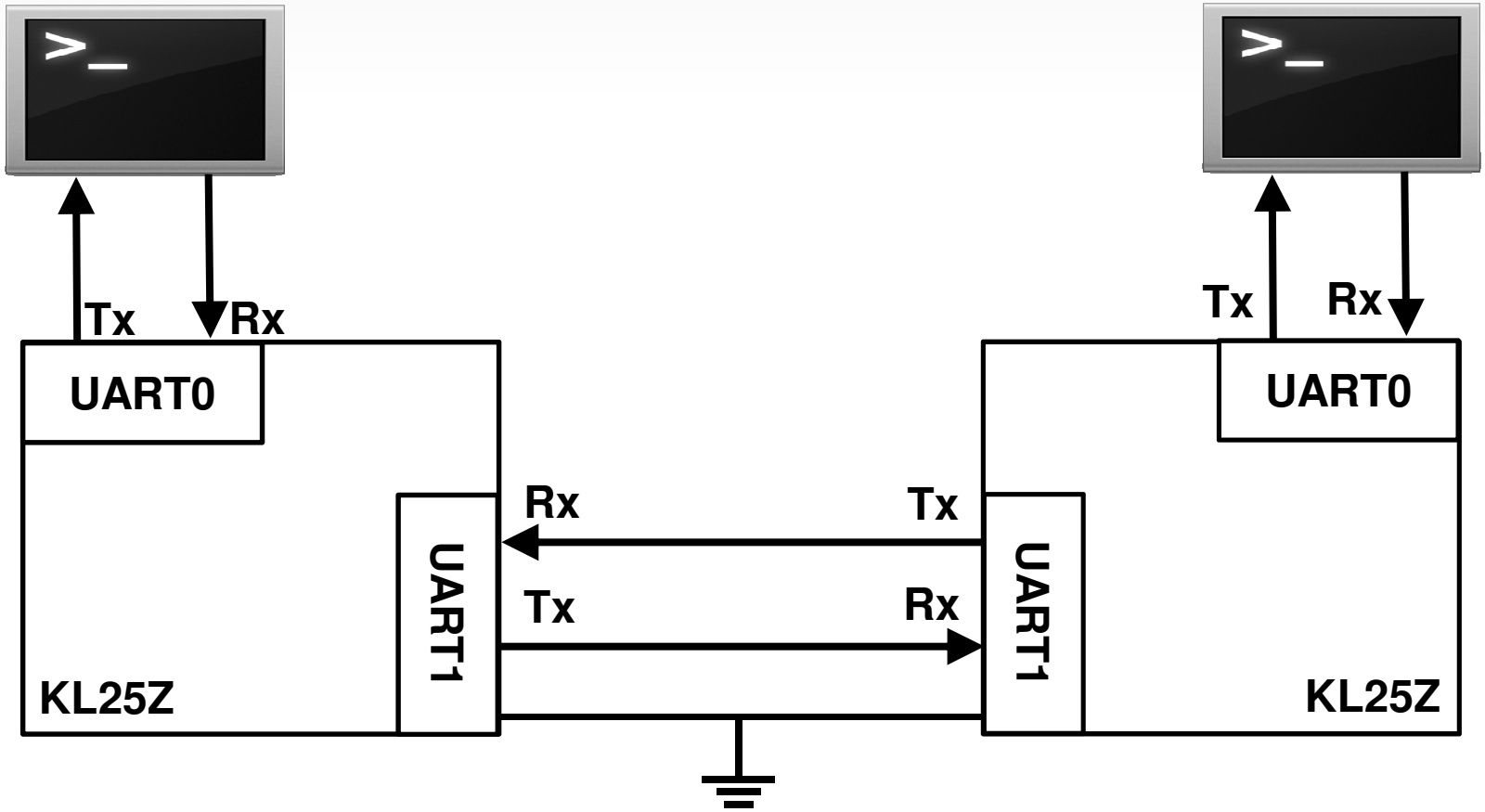
- **RAM/Stack usage:**
  - The other negative impact of using printf(): it uses a lot of stack space. This because handling all the formatters plus the variable argument list has a price. It depends on the printf()implementation, but it easily adds 512-1024 bytes on the stack. If your application causes a stack overflow, it could be because of printf().



Hands-On

- Interrupt mode:
  - In main.h file
    - `#define UART_MODE INTERRUPT`
- Polling mode:
  - In main.h file
    - `#define UART_MODE POLLING`

Hands-On




freescale™
semiconductor