بِسْمِ ٱللَّهِ ٱلرَّحْمَٰنِ ٱلرَّحِيمِ

CSE211s:

Introduction to Embedded Systems

GPIO, SysTick & UART

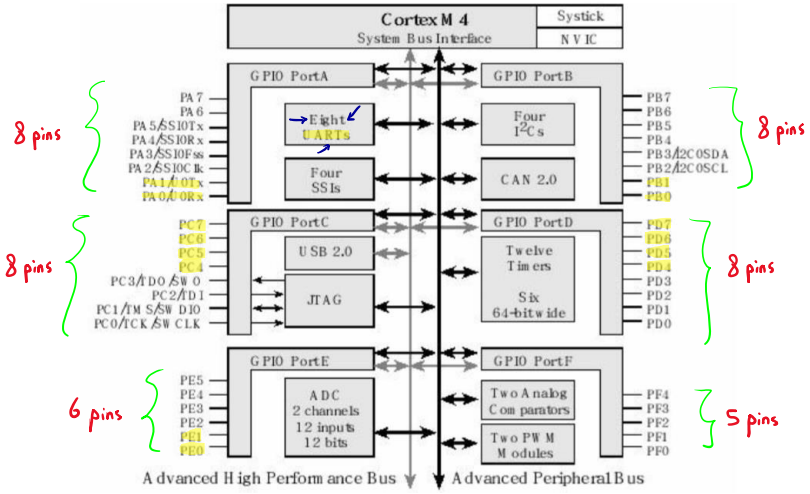
﴿يَرْفَعِ ٱللَّهُ ٱلَّذِينَ ءَامَنُوا۟ مِنكُمْ وَٱلَّذِينَ أُوتُوا۟ ٱلْعِلْمَ دَرَجَٰتٍ﴾

* General-purpose Input/Output (GPIO):

→ A parallel I/O port is a simple mechanism that allows the software to interact with external devices. It's called parallel because multiple signals can be accessed all at once.

→ The GPIO module is composed of six physical GPIO blocks, each corresponding to an individual GPIO port (Ports A, B, C, D, E, F)

→ The GPIO module supports up to 43 programmable I/O pins, depending on the peripherals being used



→ Ports accessed through the Advanced Peripheral Bus (APB)

→ Individual port pins can be general purpose I/O (GPIO) or have an alternate function

→ Microcontrollers use the concept of a direction register to determine whether a pin is an input (direction register bit is 0) or an output (direction register bit is 1)

→ The regular function of a pin is to perform parallel I/O, however, have one or more alternative functions

→ It's required to set the corresponding bit as a digital input or output

* Registers of Ports:

→ GPIO Data (GPIODATA):

- it's the data register, values written in it are transferred onto the GPIO port pins if the respective pins have been configured as outputs.
- Similarly, it returns the value on the corresponding input pin when these are configured as inputs.
- All bits are cleared by reset.

→ GPIO Direction (GPIODIR):

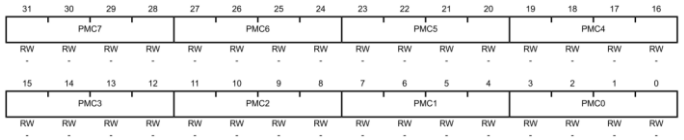
- it's the data direction register. Setting a bit in it configures the corresponding pin to be an output.
- while clearing a bit configures the corresponding pin to be an input.
- All bits are cleared by a reset (meaning all GPIO pins are inputs by default)

→ GPIO Alternate Function Selection (GPIOAFSEL):

- It's the mode control select register.
- if a bit is clear, the pin is used as a GPIO & controlled by GPIO registers
- Setting a bit, configures the corresponding GPIO line to be controlled by an associated peripheral. Several possible peripheral functions are multiplexed on each GPIO & the GPIOPCTL register is used to select one of the possible functions.
- Most bits are cleared by reset. (therefore most pins are configured as GPIOs by default)

→ GPIO Port Control (GPIOPCTL):

- It's used in conjuction with the GPIOAFSEL register & selects the specific peripheral signal for each pin when using the alternate function mode.
- It selects one out of a set of peripheral functions for each GPIO
- It's 32-bit register, Each pin has 4 bits



→ GPIO Pull-Up Select (GPIOPUR):

- It's pull-up control register. when a bit is set, a weak pull-up resistor on the corresponding GPIO is enabled.

→ GPIO Digital Enable (GPIODEN):

- It's the digital enable register. To use the pin as a digital input or output the corresponding bit must be set.

→ GPIO Lock (GPIOLOCK):

- It enables write access to the GPIOCR Register. Writing 0x4C4F.434B to the GPIOLOCK register unlocks the GPIOCR Register. Writing any other value re-enables the locked state

→ GPIO Commit (GPIOCR):

- The value of the register determines which bits of the {GPIOAFSEL, GPIOPUR, GPIODEN} registers are committed when a write to these registers is performed.
- if a bit in GPIOCR is cleared, the data being written cannot be committed.
- if a bit is set, the data being written can be committed & reflects the new value
- the contents of the GPIOCR register can only be modified if the GPIOLOCK register is unlocked. Writes is ignored if GPIOLOCK if locked.

→ GPIO Analog mode Select (GPIOAMSEL):

- it controls isolation circuits to the analog side
- if a bit is cleared, the analog function is disabled
- if a bit is Set, the analog function is enabled

→ GPIO Run Mode Clock gating Control (RCGCGPIO):

- it provides software the capability to enable & disable GPIO modules in Run mode.
- When enable (Set), a module is provided a clock & accesses to registers are allowed.
- when disabled, (Cleared) the clock is disabled to save power & accesses to registers generate a bus fault.
- All bits are cleared by reset.
- Each bit represents a port.

Address	7	6	5	4	3	2	1	0	Name
$400F.E608	-	-	GPIOF	GPIOE	GPIOD	GPIOC	GPIOB	GPIOA	SYSCTL_RCGCGPIO_R
$400F.EA08	-	-	GPIOF	GPIOE	GPIOD	GPIOC	GPIOB	GPIOA	SYSCTL_PRGPIO_R

⇒ there also a similar register specified for UART modules, Each bit from 0-7 bits represents a module. (RCGCUART)

7	6	5	4	3	2	1	0
R7	R6	R5	R4	R3	R2	R1	R0

→ GPIO Peripheral Ready (PRGPIO):

- It's a Status register which indicates whether the GPIO are ready to be accessed by software or not
- when equal 0, then the GPIO is not ready for access
- when equal 1, then the GPIO is ready for access

⇒ to use the pins for a specific peripheral, there is some steps of initializations needed for the corresponding pins & ports.

* Steps to initialize a port:

1. Activate the clock for the port by setting the corresponding bit in RCGCGPIO register & wait for PRGPIO to be true:

```
SYSCTL_RCGCGPIO_R |= 0x20;              //PORT F Clock enable
while ( (SYSCTL_PRGPIO_R&0x20) == 0) {}
```

2. Unlock the port: → Set specific value to lock register; → Set certain bits for the meant pins to CR register

```
GPIO_PORTF_LOCK_R = 0x4C4F434B;       //Unlock PORT F
GPIO_PORTF_CR_R |= 0x1F;               //Allow changes to pins 0,1,2,3,4
```

3. Set its direction register, A bit in DIR set 0 means input & 1 means output:

```
GPIO_PORTF_DIR_R |= 0x0E;              //SET pins 1,2,3 to be output
GPIO_PORTF_DIR_R &= ~0x11;             //SET pins 0,4 to be input

//Another method: GPIO_PORTF_DIR_R = 0x0E;
/*Using assign instead of logic operations in case of
  being aware of whole program and used pins*/
```

if needed as digital

4. Enable the digital port: → Set certain bits for the meant pins to DEN register

```
GPIO_PORTF_DEN_R |= 0x1F;              //SET pins 0,1,2,3,4 to be digital
```

5. Disable the analog function of the pin (As meant to be used as digital I/O):

```
GPIO_PORTF_AMSEL_R &= ~0x1F;           //Disable Analog function
```

if needed as GPIO

6. Clear bits in the alternate function register & clear bits in PCTL

```
GPIO_PORTF_AFSEL_R &= ~0x1F;           //No alternative function
GPIO_PORTF_PCTL_R &= ~0xFFFFF;         //GPIO clear bit PCTL
```

→ For alternate function: Set bits in AFSEL for corresponding pins & use the table to set bits in PCTL

7. Set the pull-up resistor if it is required (optional)

```
GPIO_PORTF_PUR_R |= 0x11;              //SET pull-up resistance
```

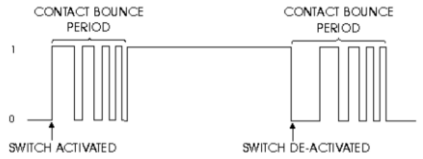
* we set the bits by doing OR operation with a mask of the meant bits also, to clear the bits by doing BIC operation with a mask of the meant bits (and operation with the Complement of the mask)

→ in Case of you are aware of the program & aware of the pins used, So you can use assign operation instead.

→ The Switches are negative logic & will require activation of the internal pull-up resistors. ⇒ in particular, you will sets 0 & 4 in GPIO_PORTF_PUR_R

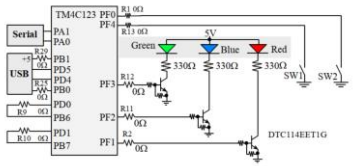
→ The LED interfaces on PF3-PF1 are positive logic, to use them make the PF3-PF1 pins an output

* Switch Debouncing: Mechanical switches may bounce when changing state → we debounce the Switch using time delay



* TivaC Lauchpad has :

- two build-in Switches
  - SW1 (PF4)
  - SW2 (PF0)
- Three LEDS
  - Red (PF1)
  - Blue (PF2)
  - Green (PF3)



* Systick Timer :

→ it's a simple Counter that we can use to create time delays & generate periodic interrupts.

→ The basis of Systick is a 24-bit down Counter that runs at the bus clock frequency.

→ The microcontroller run at 80MHz, then the SysTick Counter decrements every 12.5ns

Address	31-24	23-17	16	15-3	2	1	0	Name
$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	NVIC_ST_CTRL_R
$E000E014	0	24-bit RELOAD value						NVIC_ST_RELOAD_R
$E000E018	0	24-bit CURRENT value of SysTick counter						NVIC_ST_CURRENT_R

* Systick Registers :

- NVIC_ST_CTRL_R → it's the Control register
  - ① Enable bit: Enables the Counter → 0 = Counter disabled, 1 = Counter enabled
  - ② interrupt bit: it enables the systick interrupt. That means, whenever the Counter reaches Zero from a reload value, the systick timer requests the interrupt Signal
  - ③ Clock source bit: indicates the Clock source → 0 = External clock, 1 = processor clock
  - ④ Count Flag: Returns 1 if timer counted to Zero
    - the Count Flag could be configured to trigger an interrupt
- NVIC_ST_RELOAD_R → the reload value can be any value in the range 0x00000001 to 0x00FFFFFF (24-bit value)

to generate a multi-short timer with a period of N processor clock cycles, use a RELOAD value of N-1

- NVIC_ST_CURRENT_R → the CURRENT is loaded with RELOAD value
  - In this way, the Systick Counter CURRENT is Continuously decrementing

* Steps to initialize the Systick :

① Clear the ENABLE bit to turn off Systick during initialization :

```
NVIC_ST_CTRL_R = 0x0;                    //Clear the ENABLE bit
```

② Set the RELOAD register with required period :

```
NVIC_ST_RELOAD_R = 0x00FFFFFF;           //The Maximum Reload Value
```

→ For the period (t) & bus clock frequency (f), then set the RELOAD with a value equal to " t*f - 1 "

```
NVIC_ST_RELOAD_R = 79999;                //1ms*80MHz - 1 = 80000 - 1
```

③ Write to the NVIC_ST_CURRENT_R value to clear the Counter

```
NVIC_ST_CURRENT_R = 0x0;
```

④ Set the ENABLE bit & Set CLK_SRC

```
NVIC_ST_CTRL_R = 0x05;                   //SET ENABLE bit to 1 to turn on Systick
                                         //SET CLK_SRC bit
```

→ by clearing INTEN bit, then we have to check for count bit

```
while( (NVIC_ST_CTRL_R & 0x00010000) == 0);      //Checking for count bit
```

*UART:

→ It's the Universal Asynchronous Receiver/Transmitter

→ This serial port allows the microcontroller to communicate with devices

Serial transmission involves sending one bit at a time, such that the data is spread out over time

→ The total no. of bits transmitted per second is called the baud rate

→ The reciprocal of the baud rate is the bit time (time to send one bit)

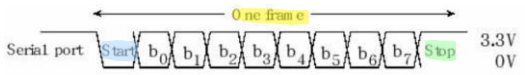
→ Each UART will have a baud rate control register, to select the transmission rate

→ There is always only one start bit, but Tiva UARTs allow us to select the 5 to 8 data bits & 1 or 2 stop bits. The UART can add even, odd, or no parity bit.

⇒ the typical protocol: 1 start bit, 8 data bits, no parity & 1 stop bit

we see that 10 bits are sent for every byte of usual data

• frame: it's the smallest complete unit of serial transmission



Showing a single frame, which includes a start bit (which is 0), 8 bits of data (least Significant bit first), and a stop bit (which is 1)

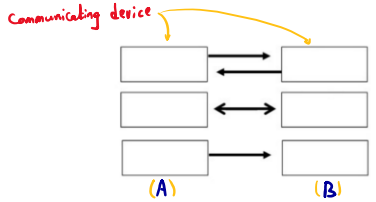
→ Serial Communication Consists of { Transmit Data, Receive Data, Signal Ground } which implements duplex Communication link

Our Tiva

① Full duplex: Used to describe any Communication that takes place in both directions at the same time. (you can both send & receive information at same time)

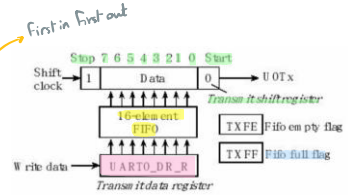
② Half duplex: It's one way Communication, used to describe any Communication that takes place in both directions but not at the same time

③ Simplex: when data transmission Can only take place in one direction



#Both Communicating devices must operate at the same bit rate

*UART Transmitter:



→ the transmitter has a 16-element fifo & a 10-bit shift register, which can not be directly accessed by the programmer. to output data using the UART, the Software will first check to make Sure the transmit fifo is not full (it will wait if TXFF is 1) & then write to the transmit data register (eg., UART0_DR_R). the bits are shifted out in this order:

Start, b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, Stop

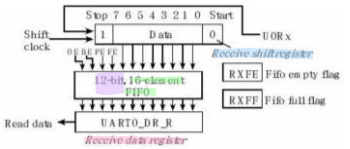
Start bit — LSB — MSB — Stop bit

8 bit data = 10 bit

When a new byte is written to UART0_DR_R, it's put into the transmit fifo. Byte by byte, the UART gets data from the fifo & loads them into the 10-bit transmit Shift register.

Then, the frame is shifted out one bit at a time at a rate specified by the baud rate register. if there are already data in the fifo or in the Shift register when the UART0_DR_R is written, the new frame will wait until the previous frames have been transmitted.

*UART Receiver:



→ The receiver cannot start a transmission but it recognizes a new frame by its start bit. the bits are shifted in using the same order as the transmitter shifted them out.

→ There are six status bits generated by receiver activity.

The Receive Fifo empty flag, RXFE, is clear when new input data are in the receive fifo. When the Software reads from UART0_DR_R, data are removed from the fifo. When the fifo becomes empty, the RXFE flag will be set, meaning there are no more input data.

→ There are other flags associated with the receiver. There is a Receive fifo Full flag RXFF, which is set when the fifo is full.

→ There are four status bits associated with each byte of data.

① overrun error (OE) ② break error (BE) ③ parity error (PE) ④ framing error (FE)

For this reason, the receive fifo is 12 bits wide

* Control bits of fifo entries :

① **Overrun error** (OE bit) → is set when the fifo is full & more input frames are arriving at the receiver which means input data are lost.

② **Parity error** (PE bit) → is set on a parity error, while creating a frame, the sender counts the number of 1s in it & adds the parity bit

→ in Case of even parity { if no. of 1s is even, PE is 0 ; if no. of 1s is odd, PE is 1

→ in Case of odd parity { if no. of 1s is odd, PE is 0 ; if no. of 1s is even, PE is 1

Most systems don't implement parity (due to error rate is low)

③ **Break error** (BE bit) → is set when the input is held low for more than a frame.

↳ Tx Signal

④ **Framing error** (FE bit) → is set when the stop bit is incorrect.

framing errors are probably caused by a mismatch in baud rate.

* UART Registers :

→ UART Data (UARTDR) :

- it's the data register (the interface to the fifos)
- for transmitted data, if the fifo is enabled, data written to this location is pushed onto the transmit fifo.
- for received data, if the fifo is enabled, the data byte & the 4-bit status is pushed onto the 12-bit wide receive fifo

BE break, FE frame, PE parity, OE overrun

	31–12	11	10	9	8	7–0	Name
$4000.C000		OE	BE	PE	FE	DATA	UART0_DR_R

- All bits are cleared by reset.

→ UART Receive Status (UARTRSR) :

- it's the receive status register. As the receive status can also be read from it as well as UARTDR

	31–3	3	2	1	0	
$4000.C004		OE	BE	PE	FE	UART0_RSR_R

- All bits are cleared by reset.

→ UART Flag (UARTFR) :

- It's the Flag register.
- After reset, TXFF, RXFF & BUSY bits are 0 & TXFE & RXFE bits are 1

	31–8	7	6	5	4	3	2–0	
$4000.C018		TXFE	RXFF	TXFF	RXFE	BUSY		UART0_FR_R

- **UART Busy** (BUSY) :
  - = 0 (the UART is not busy)
  - = 1 (the UART is busy transmitting data)

  ↳ This bit remains set until the complete byte, including all stop bits, has been sent from the Shift register

- **UART Receive Fifo empty** (RXFE) :
  - = 0 (The receiver is not empty)
  - = 1 (the receive Fifo is empty) — if the fifo is enable (FEN is 1)

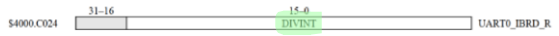
- **UART Transmit Fifo Full** (TXFF) :
  - = 0 (The transmitter is not full)
  - = 1 (the transmit Fifo is full) — if the fifo is enable (FEN is 1)

- **UART Receive Fifo Full** (RXFF) :
  - = 0 (The receiver can receive data)
  - = 1 (the receive Fifo is full) — if the fifo is enable (FEN is 1)

- **UART Transmit Fifo empty** (TXFE) :
  - = 0 (The transmitter has data to transmit)
  - = 1 (the transmit Fifo is empty) — if the fifo is enable (FEN is 1)

→ UART Integer Baud-Rate Divisor (UARTIBRD):

- It's holding the integer part of the baud rate divisor
- These bits are cleared by reset.

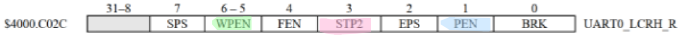


→ UART Fractional Baud-Rate Divisor (UARTFBRD):

- It's holding the Fractional part of the baud rate divisor
- These bits are cleared by reset.

→ UART Line Control (UARTLCRH):

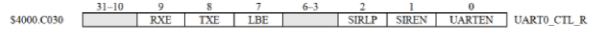
- It's the line Control register. Serial paramters such as data length, parity & stop bit selection are implemented in this register



- Break (BRK): = 0 (For normal use) [Protocal] / = 1 (Sent break)
- Parity Enable (PEN): = 0 (Parity is disabled & no parity bit added to the data frame) [Protocal] / = 1 (parity Checking & generation is enabled)
- Even parity Select (EPS): = 0 (Odd parity) / = 1 (even parity)
  - This bit has no effect when PEN bit disables parity checking & generation
- Two Stop bits Select (STP2): = 0 (One Stop bit) [Protocal] / = 1 (two Stop bits)
- FiFo Enable (FEN): = 0 (the FiFos are disabled) / = 1 (transmit & receive fifo buffers are enabled)
- Word length (WLEN): = 11 (8 bits) [Protocal] / = 10 (7 bits) / = 01 (6 bits) / = 00 (5 bits)
- Stick parity Select (SPS): = 0 (Stick parity is disabled) [Protocal] / = 1 (depends on EPS)
  - This bit has no effect when PEN bit disables parity checking & generation

→ UART Control (UARTCTL):

- It's the Control register used to enable/disable the UART
- After reset, All bits are cleared & TXE & RXE bits are 1



- UART enable (UARTEN): = 0 (UART is disabled) / = 1 (UART is enabled)
- Transmit enable (TXE): = 0 (transmit section is disabled) / = 1 (transmit section is enabled)
- Receive enable (RXE): = 0 (Receive section is disabled) / = 1 (Receive section is enabled)

⇒ Note
- to enable transmission → UARTEN & TXE must be set to 1
- to enable reception → UARTEN & RXE must be set to 1

* UART Baud Rate Generation:

→ $BRD = \frac{\text{SysClk}}{\text{ClkDiv} * \text{Baud Rate}} = \text{IBRD} + \text{FBRD} = x.y$

(SysClk default: 16 MHz; ClkDiv default: 16; IBRD = x; FBRD = 0.y)

→ $DIVINT = INT(BRD)$ (= x)

→ $DIVFRAC = INT(FBRD * 64 + 0.5)$ (FBRD = 0.y)

→ total time = total no. of bits * bit time = $\frac{\text{total no. of bits}}{\text{baud Rate}}$

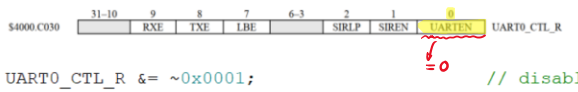
* Steps to initialize a UART0 :

1. Activate both Clocks of the UART & the port including it by setting the corresponding bits in RCGCGPIO & RCGCUART registers & wait for PRGPIO to be true :

```
SYSCTL_RCGCUART_R |= 0x0001;              // activate UART0
SYSCTL_RCGCGPIO_R |= 0x0001;              // activate port A

while ( (SYSCTL_PRGPIO_R&0x0001) == 0) {}   // checking for clock to be activated
```

2. Disable the UART by clearing the corresponding bit in UART0_CTL_R



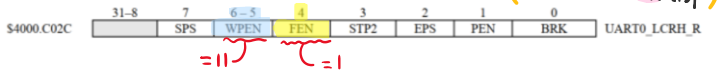

```
UART0_CTL_R &= ~0x0001;                 // disable UART
```

3. Write the values For baud Rate in UARTIBRD & UARTFBRD

```
UART0_IBRD_R = 520;          //IBRD = int(80000000/(16*9600)) = int(520.83333)
UART0_FBRD_R = 53;           //FBRD = int(0.83333 * 64 + 0.5)
```

4. Write the desired parameters — word length, number of stop bits, FIFO enable, parity enable & type in UART0_LCRH_R

Enable FiFo & 8-bit word length (Remember { no parity, one stop)




```
UART0_LCRH_R = 0x0070;                  //8-bit word length, enable FIFO
```

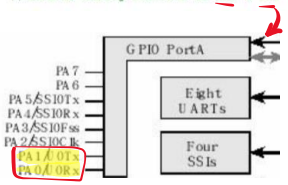
5. Enable RXE, TXE & UART by Setting the corresponding bit in UART0_CTL_R


```
UART0_CTL_R = 0x0301;                   //enable RXE, TXE & UART
```

6. Enable Alternate function, Set the corresponding bit in PCTL, enable digital I/O & disable analog function (Set GPIO AFSEL, PCTL & DEN)

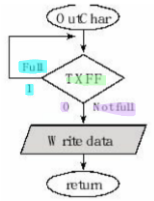
Set ox1 for PA0
Set ox1 for PA1

```
GPIO_PORTA_AFSEL_R |= 0x03;          //enable alt function PA0(U0Rx) & PA1(U0Tx)
GPIO_PORTA_PCTL_R = (GPIO_PORTA_PCTL_R&0xFFFFFF00)+0x00000011;   //Using UART Functionality for PA0 & PA1
GPIO_PORTA_DEN_R |= 0x03;            //enable digital I/O on PA0 & PA1
GPIO_PORTA_AMSEL_R  &= ~0x03;        //disable analog function on PA0 & PA1
```



* Steps to Transmit One byte :

1. Check for the transmit fifo is it full or not by checking the TXFF flag in UART0_FR_R



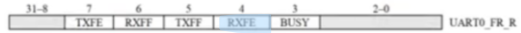
While the transmit fifo is full do nothing, when it's not Full write data

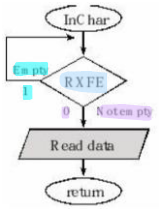
2. Assign the data to UART0_DR_R

```
void UART0_Transmit (char data)
{
  while ( (UART0_FR_R & 0x20) != 0 ) {};
  UART0_DR_R = data;
}
```

* Steps to Receive One byte :

1. Check whether there data to be received or not by checking the RXFE flag in UART0_FR_R





while the receive fifo is empty then there no data to be received thus do nothing, when it's not empty, then read the data

2. Return byte of data written in UART0_DR_R

```
char UART0_Read(void)
{
  while ( (UART0_FR_R & 0x10) != 0 );
  return (char)(UART0_DR_R & 0xFF);      //return the first 8 bits (data)
}
```

we Can also Check For Flags by using boolean function :

```
#include "stdbool.h"
bool UART0_InChar(void)
{
  return ((UART0_FR_R & 0x10) != 0) ? 0:1;     //whether there data to receive or not
}

bool UART0_OutChar (void)
{
  return ((UART0_FR_R & 0x20) != 0) ? 0:1;     //whether the buffer is full or not
}
```

Return 0 if the condition is true

Return 1 if the condition is false

only returns 1 or 0

* Test yourself → QUIZ