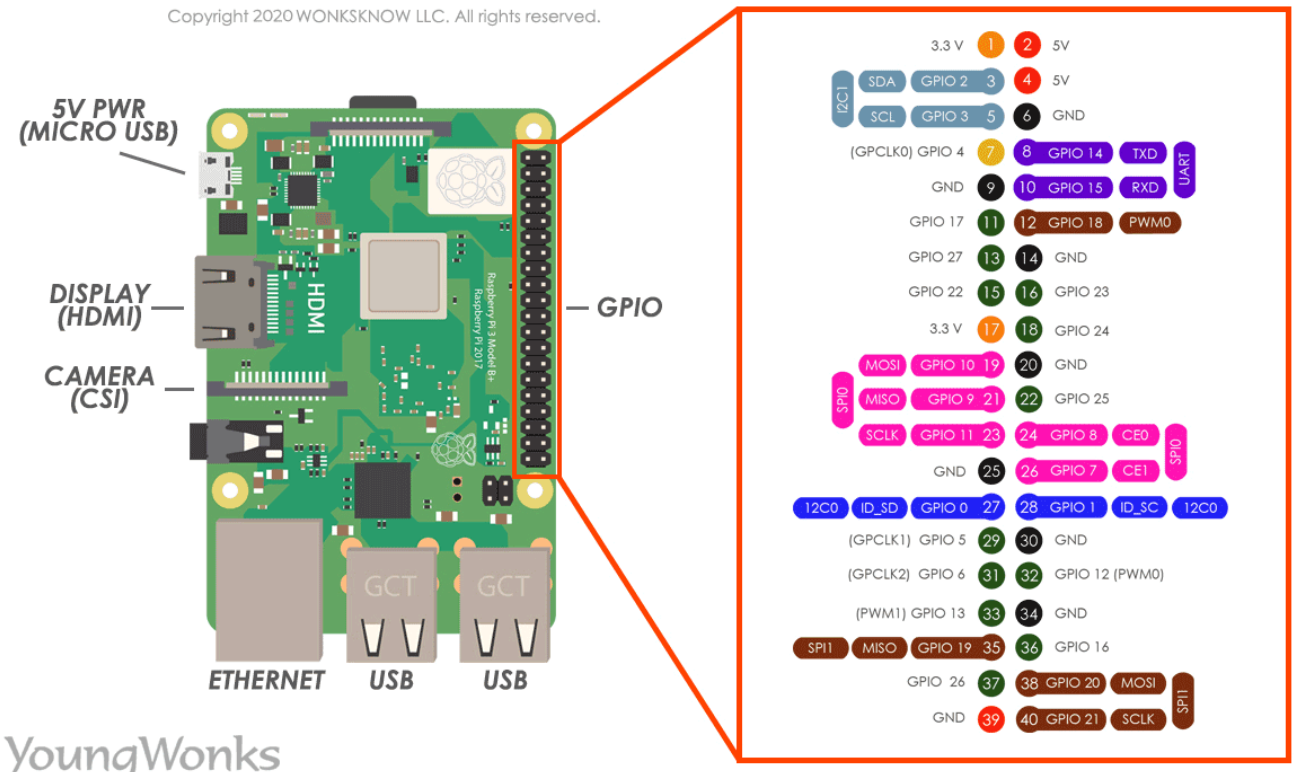
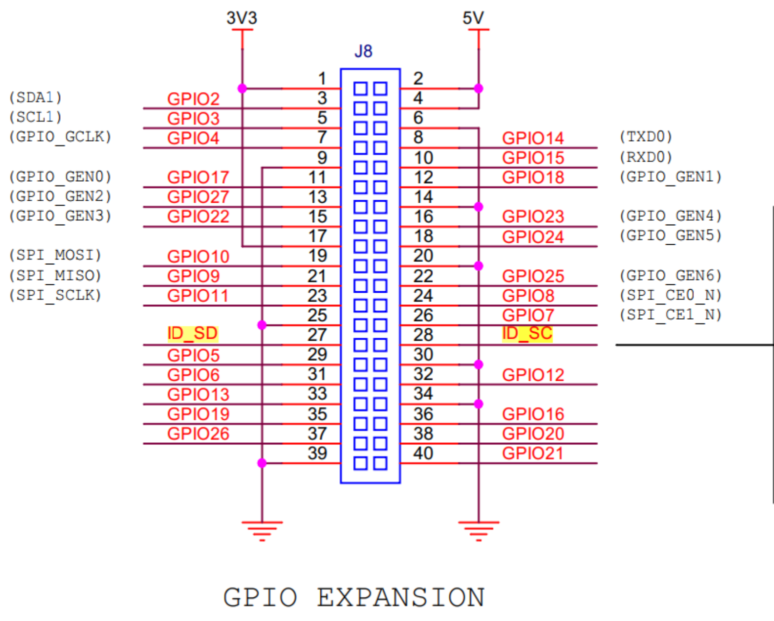
Raspberry Pi 3/4 Model B GPIO PinOut



GPIO Connection on Board’s Schematic of RPI3/4:



ID_SD and ID_SC PINS:

These pins are reserved for HAT ID EEPROM.

At boot time this I2C interface will be interrogated to look for an EEPROM that identifes the attached board and allows automagic setup of the GPIOs (and optionally, Linux drivers).

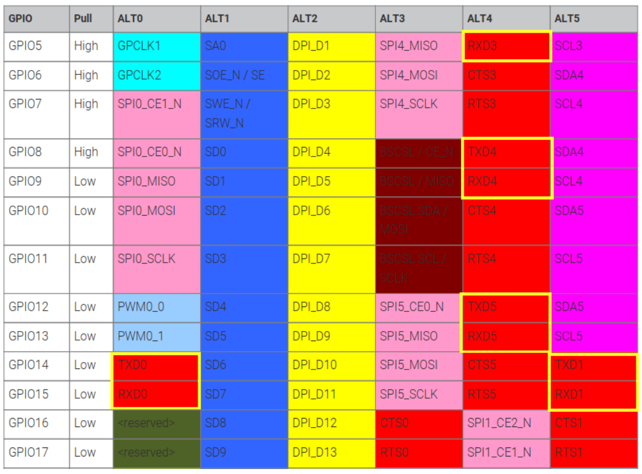
DO NOT USE these pins for anything other than attaching an I2C ID EEPROM. Leave unconnected if ID EEPROM not required.

Alternative Function Assignments

Every GPIO pin can carry an alternate function (up to 6 different functions).

For example, on RPI4, the function of GPIO5 pin can be selected among GPCLK1, SA0, DPI_D1, SPI4_MISO, RXD3, SCL3 as in the figure.

*You can refer to the **ARM Peripherals Guide (RPI3/RPI4)** of the board for detailed alternate functions of all GPIO pins.*



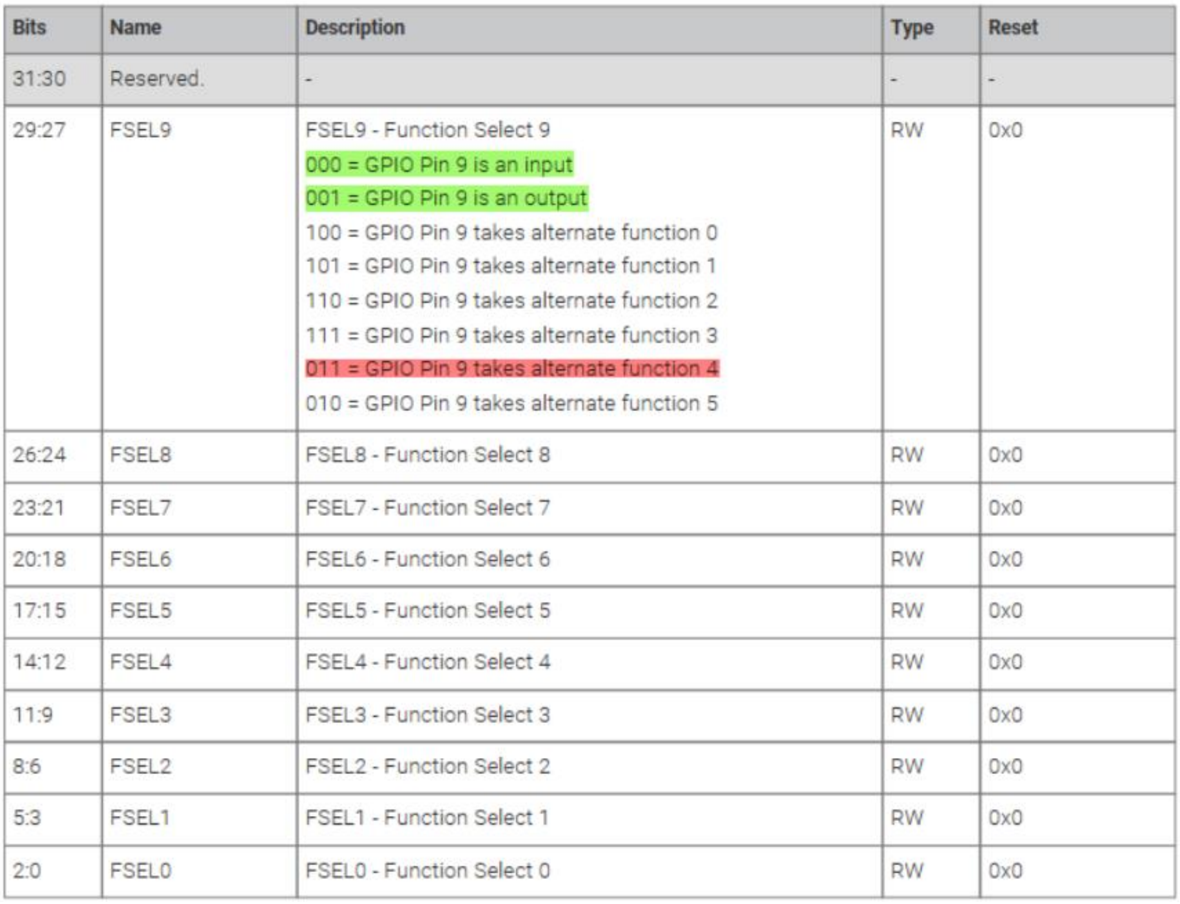
GPIO	Pull	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5
GPIO5	High	GPCLK1	SA0	DPI_D1	SPI4_MISO	RXD3	SCL3
GPIO6	High	GPCLK2	SOE_N / SE	DPI_D2	SPI4_MOSI	CTS3	SDA4
GPIO7	High	SPI0_CE1_N	SWE_N / SRW_N	DPI_D3	SPI4_SCLK	RTS3	SCL4
GPIO8	High	SPI0_CE0_N	SD0	DPI_D4	BSCSL / CE_N	TXD4	SDA4
GPIO9	Low	SPI0_MISO	SD1	DPI_D5	BSCSL / MISO	RXD4	SCL4
GPIO10	Low	SPI0_MOSI	SD2	DPI_D6	BSCSL SDA / MOSI	CTS4	SDA5
GPIO11	Low	SPI0_SCLK	SD3	DPI_D7	BSCSL SCL / SCLK	RTS4	SCL5
GPIO12	Low	PWM0_0	SD4	DPI_D8	SPI5_CE0_N	TXD5	SDA5
GPIO13	Low	PWM0_1	SD5	DPI_D9	SPI5_MISO	RXD5	SCL5
GPIO14	Low	TXD0	SD6	DPI_D10	SPI5_MOSI	CTS5	TXD1
GPIO15	Low	RXD0	SD7	DPI_D11	SPI5_SCLK	RTS5	RXD1
GPIO16	Low	<reserved>	SD8	DPI_D12	CTS0	SPI1_CE2_N	CTS1
GPIO17	Low	<reserved>	SD9	DPI_D13	RTS0	SPI1_CE1_N	RTS1

Example View of alternative functions on RPI4

To select the function for a GPIO pin, we can assign an appropriate value to a bit group in a corresponding GPFSEL register.

For example, in the RPI4, the register **GPFSEL0** is allowed to select the functions for **GPIO pins 9-0** through the **FSEL9-FSEL0** fields. Note that the detailed description given for FSEL9 can be applied for all other fields. Therefore, if you want to configure the **GPIO5** as **RXD3** (**ATL4** function as in the table above), you need to set the **FSEL5 (bits 17:15) of GPFSEL0** to **binary value 011.**

GPFSEL0 Register



Bits	Name	Description	Type	Reset
31:30	Reserved.	-	-	-
29:27	FSEL9	FSEL9 - Function Select 9 000 = GPIO Pin 9 is an input 001 = GPIO Pin 9 is an output 100 = GPIO Pin 9 takes alternate function 0 101 = GPIO Pin 9 takes alternate function 1 110 = GPIO Pin 9 takes alternate function 2 111 = GPIO Pin 9 takes alternate function 3 011 = GPIO Pin 9 takes alternate function 4 010 = GPIO Pin 9 takes alternate function 5	RW	0x0
26:24	FSEL8	FSEL8 - Function Select 8	RW	0x0
23:21	FSEL7	FSEL7 - Function Select 7	RW	0x0
20:18	FSEL6	FSEL6 - Function Select 6	RW	0x0
17:15	FSEL5	FSEL5 - Function Select 5	RW	0x0
14:12	FSEL4	FSEL4 - Function Select 4	RW	0x0
11:9	FSEL3	FSEL3 - Function Select 3	RW	0x0
8:6	FSEL2	FSEL2 - Function Select 2	RW	0x0
5:3	FSEL1	FSEL1 - Function Select 1	RW	0x0
2:0	FSEL0	FSEL0 - Function Select 0	RW	0x0

Peripheral Base Address

To configure a peripheral, we need to know its registers' addresses.

For RPI4, the **ARM Peripherals** document mention that the peripheral addresses are described in the document as being **at legacy address 0x7Enn_nnnn,** available in the 35-bit address space at 0x4_7Enn_nnnn, and **visible to the ARM (CPU) at 0x0_FEnn_nnnn if Low Peripheral mode is enabled**.

*Since **RPi4** boots into Low Peripheral mode by default, the **registers' addresses are started at base address 0xFE000000** (instead of legacy address 0x7E000000 in the document)*. That's why we define `MMIO_BASE` value is `0xFE000000` for RPI4 in our **gpio.h** header file.

```
#define MMIO_BASE       0xFE000000
```

For RPI3, the **ARM Peripherals** document discloses that "physical addresses range from 0x3F000000 to 0x3FFFFFFF for peripherals. A peripheral advertised in the document at bus address 0x7Ennnnnn is available at physical address 0x3Fnnnnnn". Therefore, for RPI3 the registers' addresses are started at **base address *0x3F000000*** *(instead of bus address 0x7E000000 in the document)*. That's also why we define `MMIO_BASE` value is `0x3F000000` for RPI3 in our **gpio.h** header file.

```
#define MMIO_BASE       0x3F000000
```

Configuring UARTs

There are two types of UART available on the Raspberry Pi 3/4 - PL011 and mini UART. The PL011 is a capable, broadly 16550-compatible UART, while the mini UART has a reduced feature set. *We will work with mini UART (UART1) first since it is easier to program.*

NOTE: All UARTs on the Raspberry Pi are 3.3V only - damage will occur if they are connected to 5V systems. An adaptor can be used to connect to 5V systems. Alternatively, low-cost USB to 3.3V serial adaptors are available from various third parties.

Name	Type	Board
UART0	PL011	RPI3/4
UART1	**mini UART**	RPI3/4
UART2	PL011	RPI4 only
UART3	PL011	RPI4 only
UART4	PL011	RPI4 only
UART5	PL011	RPI4 only

Mini UART (UART1)

1. Procedure to initialize UART1

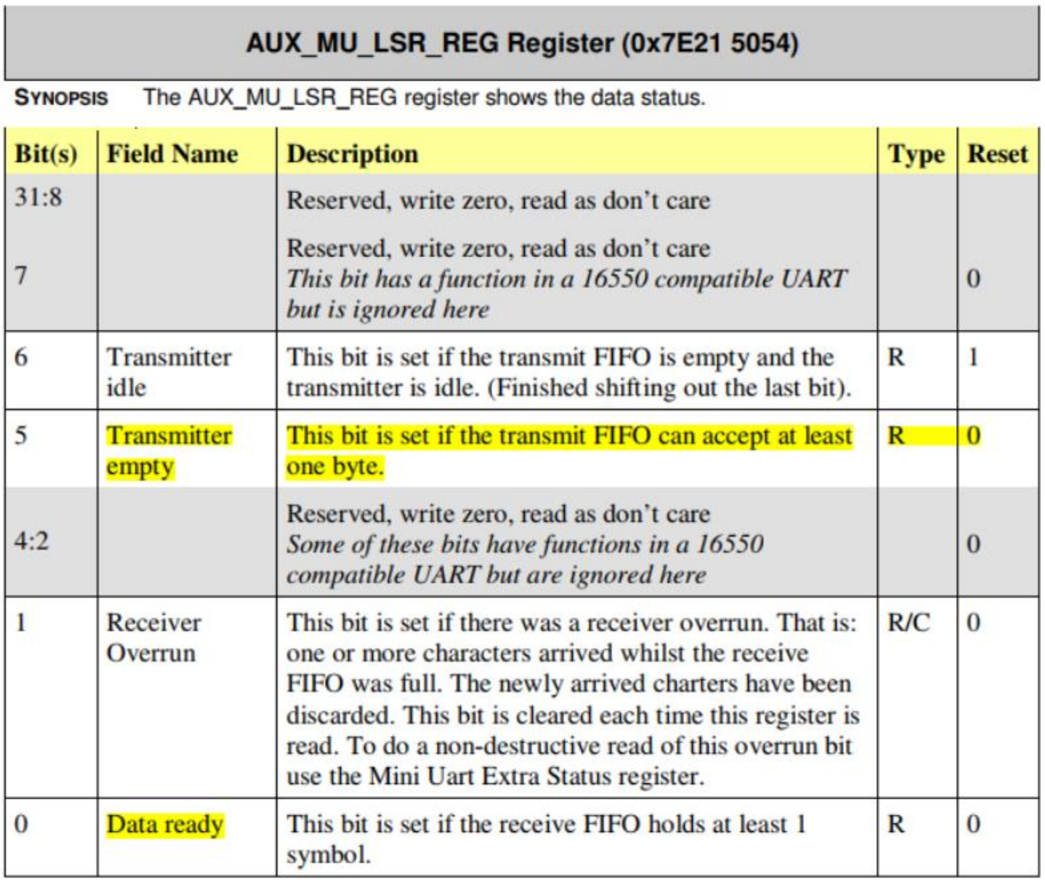
- Enable UART1 in **AUXENB** register.
- Configure **AUX_MU_CNTL** register to stop transmitter and receiver (TX and RX)
- Set operating mode (e.g., 8 bits, no interrupt, 115200 baud) through **AUX_MU_LCR, AUX_MU_MCR, AUX_MU_IER, AUX_MU_BAUD** registers.
- Clear data buffers by writing to **AUX_MU_IIR** register.
- Map UART1 (TXD1, RXD1) to appropriate GPIO pins, e.g. pins 14 and 15, through GPIO alternative function selection (FSEL fields in a **GPFSEL register**).
- Set no pull up/ pull down control, and enable clock for the selected GPIO pins through **GPPUD** and **GPPUDCLK** registers.
- Enable UART1's transmitter and receiver (TX and RX) in **AUX_MU_CNTL** register

Example Code:

```
void uart_init()
{
   register unsigned int r;

   /* initialize UART */
   AUX_ENABLE |= 1;     //enable UART1, AUX mini uart (AUXENB)
   AUX_MU_CNTL = 0;           //stop transmitter and receiver
   AUX_MU_LCR  = 3;    //8-bit mode (also enable bit 1 to be used for RBP3)
   AUX_MU_MCR  = 0;           //RTS (request to send)
   AUX_MU_IER  = 0;   //disable interrupts
   AUX_MU_IIR  = 0xC6;  //clear FIFOs
   AUX_MU_BAUD = 270;   // 115200 baud   ( system_clk_freq/(baud_rate*8) – 1 )

   /* map UART1 to GPIO pins */
   r = GPFSEL1;
   r &= ~( (7 << 12)|(7 << 15) ); //Clear bits 12-17 (gpio14, gpio15)
   r |=    (2 << 12)|(2 << 15);   //Set value 2 (select ALT5: UART1)
   GPFSEL1 = r;

   /* enable GPIO 14, 15 */
   GPPUD = 0;           //No pull up/down control
   r = 150;  while(r--) { asm volatile("nop"); } //waiting 150 cycles
   GPPUDCLK0 = (1 << 14)|(1 << 15); //enable clock for GPIO 14, 15
   r = 150;  while(r--) { asm volatile("nop"); } //waiting 150 cycles
   GPPUDCLK0 = 0;       // flush GPIO setup

   AUX_MU_CNTL = 3;     //Enable transmitter and receiver (Tx, Rx)
}
```

2. Send/Receive character from UART1

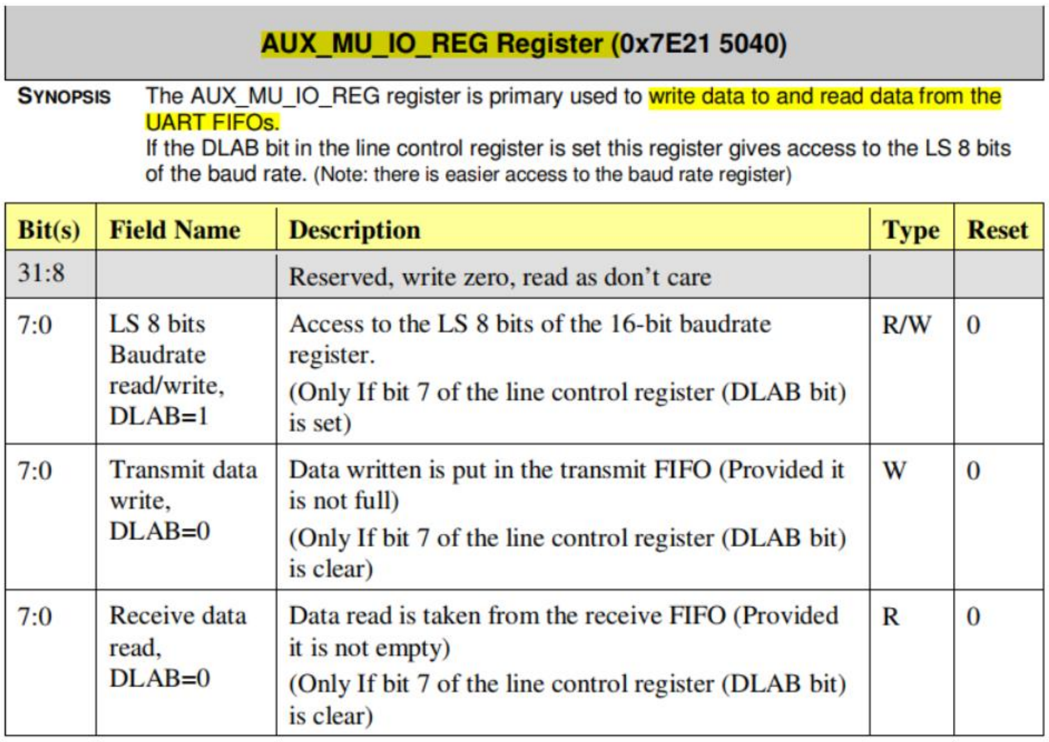
To send/receive data from UART1, we need to refer to two registers AUX_MU_LSR and AUX_MU_IO as below:



AUX_MU_LSR_REG Register (0x7E21 5054)

SYNOPSIS The AUX_MU_LSR_REG register shows the data status.

Bit(s)	Field Name	Description	Type	Reset
31:8		Reserved, write zero, read as don't care		
7		Reserved, write zero, read as don't care *This bit has a function in a 16550 compatible UART but is ignored here*		0
6	Transmitter idle	This bit is set if the transmit FIFO is empty and the transmitter is idle. (Finished shifting out the last bit).	R	1
5	Transmitter empty	This bit is set if the transmit FIFO can accept at least one byte.	R	0
4:2		Reserved, write zero, read as don't care *Some of these bits have functions in a 16550 compatible UART but are ignored here*		0
1	Receiver Overrun	This bit is set if there was a receiver overrun. That is: one or more characters arrived whilst the receive FIFO was full. The newly arrived charters have been discarded. This bit is cleared each time this register is read. To do a non-destructive read of this overrun bit use the Mini Uart Extra Status register.	R/C	0
0	Data ready	This bit is set if the receive FIFO holds at least 1 symbol.	R	0



AUX_MU_IO_REG Register (0x7E21 5040)

SYNOPSIS The AUX_MU_IO_REG register is primary used to write data to and read data from the UART FIFOs.
If the DLAB bit in the line control register is set this register gives access to the LS 8 bits of the baud rate. (Note: there is easier access to the baud rate register)

Bit(s)	Field Name	Description	Type	Reset
31:8		Reserved, write zero, read as don't care		
7:0	LS 8 bits Baudrate read/write, DLAB=1	Access to the LS 8 bits of the 16-bit baudrate register. (Only If bit 7 of the line control register (DLAB bit) is set)	R/W	0
7:0	Transmit data write, DLAB=0	Data written is put in the transmit FIFO (Provided it is not full) (Only If bit 7 of the line control register (DLAB bit) is clear)	W	0
7:0	Receive data read, DLAB=0	Data read is taken from the receive FIFO (Provided it is not empty) (Only If bit 7 of the line control register (DLAB bit) is clear)	R	0

Generally, to send a character (8 bits) through UART1, we need to check and wait until bit 5 (transmitter empty) of the AUX_MU_LSR is asserted. Then we can simply write the data to AUX_MU_IO to be sent out.

Example Code:

```c
/**
 * Send a character
 */
void uart_sendc(unsigned char c) {
    /* wait until transmitter is empty */
    do {
        asm volatile("nop");
    } while ( !(AUX_MU_LSR & 0x20) );

    /* write the character to the buffer */
    AUX_MU_IO = c;
}
```

Similarly, to get a character (8 bits) through UART1, we need to check and wait until bit 0 (data ready) of the AUX_MU_LSR is asserted. Then we can simply read the data from the AUX_MU_IO register.

Example Code:

```c
/**
 * Receive a character
 */
char uart_getc() {
    char c;

    /* wait until data is ready (one symbol) */
    do {
        asm volatile("nop");
    } while ( !(AUX_MU_LSR & 0x01) );

    /* read it and return */
    c = (char)(AUX_MU_IO);

    /* convert carriage return to newline */
    return (c == '\r' ? '\n' : c);
}
```

3. Mini-UART and CPU Core Frequency

In order to use the mini UART, you also need to configure the Raspberry Pi to use a fixed VPU core clock frequency. This is because the mini UART clock is linked to the VPU core clock, so that when the core clock frequency changes, the UART baud rate will also change.

The **enable_uart** and **core_freq** settings can be added to **config.txt** file inside the microSD card to change the behaviour of the mini UART as below:

```
File: config.txt
Add two additional settings
enable_uart=1
core_freq=250
```

PL011 UART (UART0/2/3/4/5)

1. Physical base address

The PL011 UARTs are mapped onto the following legacy base addresses in the ARM Peripheral document:
• UART0: 0x7e201000
• UART2: 0x7e201400
• UART3: 0x7e201600
• UART4: 0x7e201800
• UART5: 0x7e201a00

*Since the physical peripheral base address is **0x3F000000** in RPI3, and **0xFE000000** in RPI4, we can convert these values to **physical addresses** as follows:*

	Physical base address in RPI4	**Physical base address in RPI3**
UART0	0xFE201000	0x3F201000
UART2	0xFE201400	- *(not available)*
UART3	0xFE201600	-
UART4	0xFE201800	-
UART5	0xFE201a00	-

2. PL011 UARTs' Registers

Each PL011 UART will have following registers for configuration:

Offset	**Name**	**Description**
0x00	**DR**	Data Register
0x04	RSRECR	Receive status register/error clear register
0x18	**FR**	Flag register
0x20	ILPR	not in use
0x24	**IBRD**	Integer Baud rate divisor
0x28	**FBRD**	Fractional Baud rate divisor
0x2c	**LCRH**	Line Control register
0x30	**CR**	Control register
0x34	IFLS	Interrupt FIFO Level Select Register
0x38	IMSC	Interrupt Mask Set Clear Register
0x3c	RIS	Raw Interrupt Status Register
0x40	MIS	Masked Interrupt Status Register
0x44	ICR	Interrupt Clear Register
0x48	DMACR	DMA Control Register
0x80	ITCR	Test Control register
0x84	ITIP	Integration test input reg
0x88	ITOP	Integration test output reg
0x8c	TDR	Test Data reg

3. Procedure to initialize PL011 UART

- Disable the UART through its control register **CR**.
- Map the UART (TXD and RXD) to appropriate GPIO pins through GPIO alternative function selection (FSEL fields in a **GPFSEL register**).
- Set no pull up/ pull down control, and enable clock for the selected GPIO pins through **GPPUD** and **GPPUDCLK** registers.
- For simplicity, disable interrupt though **IMSC** and **ICR** registers**.**
- Setup expected **baud rate** though **IBRD** and **FBRD** registers:

  Baud rate divisor **BAUDDIV** = (UART_CLOCK /(16 * Baud rate))

  Integer part register **UART_IBRD** = *Integer part* of **BAUDDIV**
  Fraction part register **UART_FBRD** = (*Fractional part* * 64) + 0.5
  Note: by default, **UART_CLOCK = 48MHz** on RPi3/4.

  Example**:**
  To setup 115200 baud rate: BAUDDIV = $48*10^6/(16*115200)$ = 26.04166667
  → IBRD = 26
  → FBRD = 0.04166667 * 64 + 0.5 = 3.16 ≈ 3

- Enable FIFO buffer, set operating mode (e.g., data length is 8 bits, no parity and 1 stop bit) through its line control register **LCRH.**
- Enable the UART, as well as its receive and transmit channels through its control register **CR**.

Example Code for UART3 (mapped on GPIO 4, 5):

```
void uart3_init()
{
    unsigned int r;

      /* Turn off UART3 */
      UART3_CR = 0x0;

      /* Setup GPIO 4 and 5
         Set GPIO4 and GPIO5 (in GPFSEL0) to be UART3 which is ALT4 */
      r = GPFSEL0;
      r &=  ~((7 << 12) | (7 << 15));        // Clear the fields FSEL4 and
FSEL5
      r |= (0b011 << 12)|(0b011 << 15);   //Set value 0b011 (select ALT4:
UART3)
      GPFSEL0 = r;

      /* Enable GPIO 4, 5 */
#ifdef RBP3 //RBP3
      GPPUD = 0;              //No pull up/down control
      //Toogle clock to flush GPIO setup
      r = 150; while(r--) { asm volatile("nop"); } //waiting 150 cycles
      GPPUDCLK0 = (1 << 14)|(1 << 15); //enable clock for GPIO 14, 15
      r = 150; while(r--) { asm volatile("nop"); } //waiting 150 cycles
      GPPUDCLK0 = 0;          // flush GPIO setup

#else //RPB4
      r = GPIO_PUP_PDN_CNTRL_REG0;
      r &= ~((3 << 8) | (3 << 10)); // Clear Resistor Select for GPIO 04, 05
      GPIO_PUP_PDN_CNTRL_REG0 = r;
#endif
```

```c
        /* Mask all interrupts. Clear all the interrupts */
        UART3_IMSC = 0;

        /* Clear pending interrupts. Clear all the interrupts */
        UART3_ICR = 0x7FF;

        /* Set integer & fractional part of Baud rate
        Divider = UART_CLOCK/(16 * Baud)
        Default UART_CLOCK = 48MHz (old firmware it was 3MHz);
        Integer part register UART3_IBRD  = integer part of Divider
        Fraction part register UART3_FBRD = (Fractional part * 64) + 0.5 */

        //115200 baud
        UART3_IBRD = 26;
        UART3_FBRD = 3;

        /* Set up the Line Control Register
        - Enable FIFO
        - Set length to 8 bit
        - Defaults for other bits. No parity, 1 stop bit */
        UART3_LCRH = UART3_LCRH_FEN | UART3_LCRH_WLEN_8BIT;

        /* Enable UART3, receive, and transmit */
        UART3_CR = 0x301;     // enable Tx, Rx, FIFO 0b0011 0000 0001
}
```

4. Send/Receive character from PL011 UART:

```c
/**
 * Send a character
 */
void uart3_sendc(unsigned char c) {

    /* Check Flags Register */
     /* And wait until transmitter is not full */
     do {
            asm volatile("nop");
     } while (*UART3_FR & UART3_FR_TXFF);

     /* Write our data byte out to the data register */
     *UART3_DR = c ;
}
```

```c
/**
 * Receive a character
 */
unsigned char uart3_getc() {
    char c = 0;

    /* Check Flags Register */
    /* Wait until Receiver is not empty
     * (at least one byte data in receive fifo)*/
     do {
           asm volatile("nop");
    } while ( *UART3_FR & UART3_FR_RXFE );

    /* read it and return */
    c = (unsigned char) (*UART3_DR);

    /* convert carriage return to newline */
    return (c == '\r' ? '\n' : c);
}
```