lib_uart: UART peripheral library

Publication Date: 2025/6/25
Document Number: XM-006381-UG v3.2.0

XMOS

IN THIS DOCUMENT

1 Inroduction

A software defined, industry-standard, UART (Universal Asynchronous Receiver/Transmitter) library that allows the user to control a UART serial connection via the xcore GPIO ports. This library is controlled via XC using the XMOS multicore extensions.

1.1 `lib_uart` components

There are four ways to use the UART library detailed in the table below.

UART type	Description
Standard	Standard UARTs provide a flexible, fully configurable UART for speeds up to 115200 baud. The UART connects to ports via the GPIO library so can be used with single bits of multi-bit ports. Transmit can be buffered or unbuffered. The UART components run on a logical core but are combinable so can be run with other tasks on the same core (though the timing may be affected).
Fast/streaming	The fast/streaming UART components provide a fixed configuration fast UART that streams data in and out via a streaming channel.
Half-duplex	The half-duplex component performs receive and transmit on the same data line. The application controls the direction of the UART at runtime. It is particularly useful for RS485 connections.
Multi-UART	The multi-UART components efficiently run several UARTS on the same core using a multibit port.

1.2 Using `lib_uart`

`lib_uart` is intended to be used with the XCommon CMake , the *XMOS* application build and dependency management system.

To use this library, include **`lib_uart`** in the application's **`APP_DEPENDENT_MODULES`** list in *CMakeLists.txt*, for example:

```
set(APP_DEPENDENT_MODULES "lib_uart")
```

Applications should then include the **`uart.h`** header file.

2 External signal description

The UART signals used by the library are high in their idle state. The transmission of a character start with a *start bit* when the line transitions from high to low. Then the data bits of the character are then transmitted followed by an optional parity bit and a number of stop bits (where the line is driven high). This sequence is shown in *UART data sequence*. The data is driven least significant bit first.

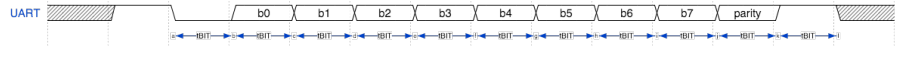

Fig. 1: UART data sequence

The start bit, data bits, parity bit and stop bits are all the same length (**`tBIT`** in *UART data sequence*). This length is give by the BAUD rate which is the number of bits per second.

2.1 Connecting to the *xcore* device

If using the standard UART Rx/Tx components then the UART line can be connected to a bit of any port. The other bits of the port can be shared using the GPIO library. Please refer to the GPIO library user guide for restrictions on sharing bits of a port (for example, all bits of a port need to be in the same direction - so UART rx and UART tx cannot be put on the same port, see *UART Rx and Tx connections*).

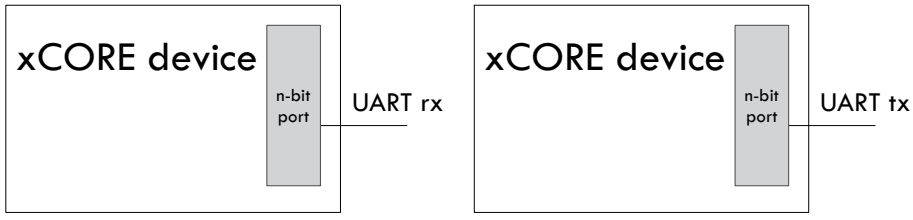

Fig. 2: UART Rx and Tx connections

The half duplex UART needs to be connected to a 1-bit port (*UART half duplex connection*).

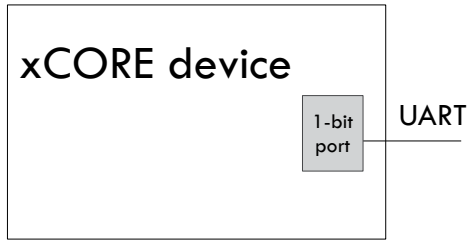

Fig. 3: UART half duplex connection

The fast/streaming UART also needs to be connect to a 1-bit port for TX or RX (*Fast/Streaming UART connections*).

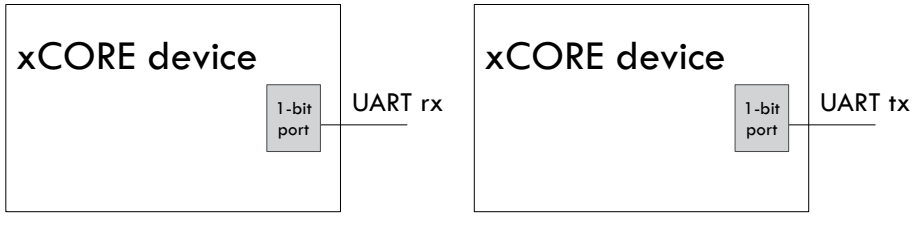

Fig. 4: Fast/Streaming UART connections

The multi-UARTs need to be connected to 8-bit ports. If fewer than 8 UARTs are required then an 8-bit port must still be used with some of the pins of the port not connected (*Multi UART connections*).

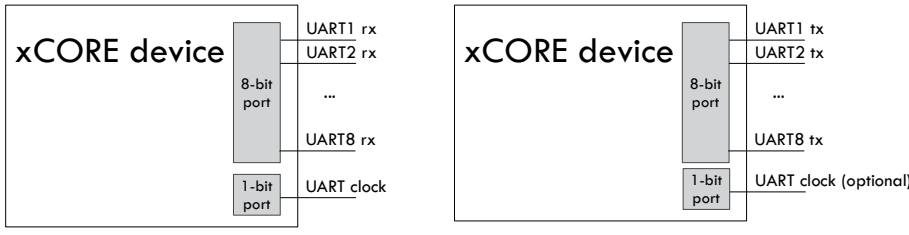

Fig. 5: Multi UART connections

For multi-UART receive, an incoming clock is required to acheive standard baud rates. The clock should be a multiple of the maximum BAUD rate required e.g. a 1843200 Hz oscillator is a multiple of 115200 baud (and lower rates also). The maximum allowable incoming signal is 1843200 Hz.

For multi-UART transmit, an incoming clock can also be used. The same clock signal can be shared between receive and transmit (i.e. only a single 1-bit port need be used).

3 Usage

The following sections describe the four ways to use the UART library.

3.1 Standard UART usage

UART components are instantiated as parallel tasks that run in a `par` statement. The application can connect via an interface connection using the `uart_rx_if` (for the UART Rx component) or the `uart_tx_if` (for the UART Tx component), see *UART task diagram* for details. Both components also have an optional configuration interface that lets the application change the speed and properties of the UART at run time.

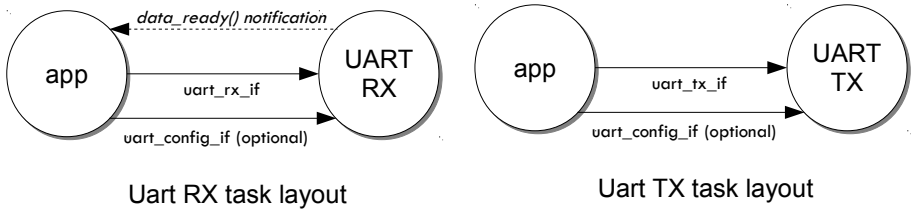

Fig. 6: UART task diagram

For example, the following code instantiates a UART rx and UART tx component and connects to them:

```
// Port declarations
port p_uart_rx = on tile[0] : XS1_PORT_1A;
port p_uart_tx = on tile[0] : XS1_PORT_1B;

#define RX_BUFFER_SIZE 20

int main() {
  interface uart_rx_if i_rx;
  interface uart_tx_if i_tx;
  input_gpio_if i_gpio_rx[1];
  output_gpio_if i_gpio_tx[1];
  par {
    on tile[0]: output_gpio(i_gpio_tx, 1, p_uart_tx, null);
    on tile[0]: uart_tx(i_tx, null,
                        115200, UART_PARITY_NONE, 8, 1,
                        i_gpio_tx[0]);
    on tile[0].core[0] : input_gpio_with_events(i_gpio_rx, 1, p_uart_rx, null);
    on tile[0].core[0] : uart_rx(i_rx, null, RX_BUFFER_SIZE,
                                 115200, UART_PARITY_NONE, 8, 1,
                                 i_gpio_rx[0]);
    on tile[0]: app(i_tx, i_rx);
  }
  return 0;
}
```

The `output_gpio` task and `input_gpio_with_events` tasks are part of the GPIO library for flexible use of multi-bit ports. See the GPIO library user guide for details.

The application can use the client end of the interface connection to perform UART operations e.g.:

```
void my_application(client uart_tx_if uart_tx,
                    client uart_rx_if uart_rx) {
   // Write a byte to the UART
   uart_tx.write(0xff);

   // Wait for a byte to
   select {
     case uart_rx.data_ready():
        uint8_t data = uart_rx.read();
        printf("Data received %d\n", data);
        // ...
        break;
   }
}
```

3.1.1 UART configuration

The `uart_config_if` connection can be optionally connected to either the UART Rx or Tx task e.g.:

```
// ...
interface uart_tx_if i_tx;
interface uart_cfg_if i_tx_cfg;
input_gpio_if i_gpio_rx[1];
par {
  // ...
  on tile[0]: uart_tx(i_tx, i_tx_cfg,
                      115200, UART_PARITY_NONE, 8, 1,
                      i_gpio_tx[0]);
  on tile[0]: app(i_tx, i_rx_cfg);
  // ...
```

The application can use this interface to dynamically reconfigure the UART e.g.:

```
void app(client uart_tx_if uart_tx,
         client uart_config_if uart_tx_cfg) {
    // Configure the UART to 9600 BAUD
    uart_tx_cfg.set_baud_rate(9600);
    // Write to the UART
    uart_tx.write(0xff);
    // ...
```

If runtime configuration is not required then `null` can be passed into the task instead of an interface connection.

3.1.2 Transmit buffering

There are two types of standard UART tx task: buffered and un-buffered.

The buffered UART will buffer characters written to the UART. It requires a separate logical core to feed characters from the buffer to the UART pin. This frees the application to perform other processing. The buffered UART will inform the application that data has been transmitted and that there is more space in the buffer by calling the *`ready_to_transmit()`* notification.

The unbuffered UART does not take its own logical core but calls to `write` will block until the character has been sent.

3.2 Fast/Streaming UART usage

The fast/streaming UART components are instantiated as parallel tasks that run in a `par` statement and connected to the application via streaming channels (*Fast/streaming UART task diagram*).

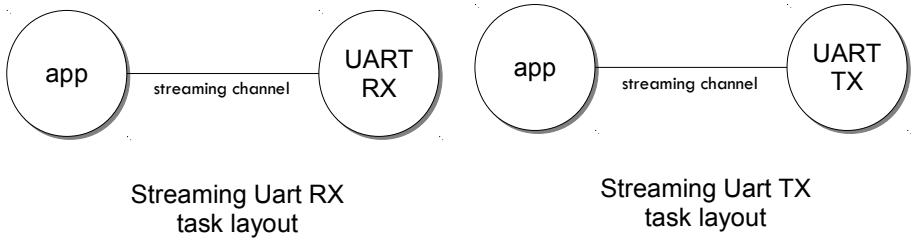

Fig. 7: Fast/streaming UART task diagram

For example, the following code instantiates a strreaming UART rx and UART tx component and connects to them:

```
// Port declarations
in port p_uart_rx = on tile[0] : XS1_PORT_1A;
out port p_uart_tx = on tile[0] : XS1_PORT_1B;

#define TICKS_PER_BIT 20

int main() {
  streaming chan c_rx;
  streaming chan c_tx;
  par {
    on tile[0]: uart_tx_streaming(p_uart_tx, c_tx, TICKS_PER_BIT);
    on tile[0]: uart_rx_streaming(p_uart_rx, c_rx, TICKS_PER_BIT);
    on tile[0]: app(c_tx, c_rx);
  }
  return 0;
}
```

The streaming channel has a limited amount of buffering (~8 characters) but in general the application must deal with incoming data as soon as it arrives.

The application can interact with the component using the fast/streaming UART functions (see *Fast/Streaming API*) e.g.:

```
void app(streaming chanend c_tx, streaming chanend c_rx)
{
   uart_tx_streaming_write_byte(c_tx, 0xff);
   uint8_t byte;
   uart_rx_streaming_read_byte(c_rx, byte);
   printf("Received: %d\n", byte);
   ...
```

3.3 Half-duplex UART usage

The half-duplex components are instantiated as parallel tasks that run in a `par` statement. The application connects via three interface connections: the `uart_rx_if` (for receiving data), the `uart_tx_if` (for transmitting data) and the `uart_control_if` (for controlling the current direction of the UART)(*Half-duplex UART task diagram*). The component also has an optional configuration interface that lets the application change the speed and properties of the UART at run time.

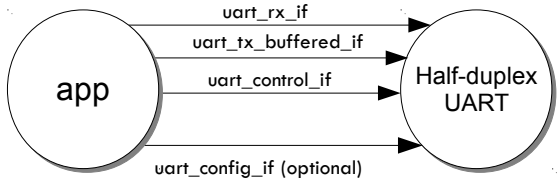

Fig. 8: Half-duplex UART task diagram

For example, the following code instantiates a half-duplex UART component and connects to it:

```
#define TX_BUFFER_SIZE 16
#define RX_BUFFER_SIZE 16

port p_uart = on tile[0] : XS1_PORT_1A;

int main() {
  interface uart_rx_if i_rx;
  interface uart_control_if i_control;
  interface uart_tx_buffered_if i_tx;

  par {
    on tile[0] : uart_half_duplex(i_tx, i_rx, i_control, null,
                                  TX_BUFFER_SIZE, RX_BUFFER_SIZE,
                                  115200, UART_PARITY_NONE, 8, 1, p_uart);

    on tile[0] : app(i_rx, i_tx, i_control);
  }
```

The application can use the interfaces in the same manner as a standard UART. The control interface can be used to change direction e.g.:

```
void app(client uart_rx_if i_uart_rx,
         client uart_tx_buffered_if i_uart_tx,
         client uart_control_if i_control) {
   uint8_t byte;
   i_control.set_mode(UART_RX_MODE);
   byte = i_uart_rx.read();
   i_control.set_mode(UART_TX_MODE);
   i_uart_tx.write(byte);
   ...
```

3.4 Multi-UART usage

Multi-UART components are instantiated as parallel tasks that run in a `par` statement. The application can connect via a combination of a channel and an interface connection using the `multi_uart_rx_if` (for the UART Rx component) or the `multi_uart_tx_if` (for the UART Tx component). These interfaces handle data for all the UARTs and runtime configuration (*Multi-UART task diagram*).

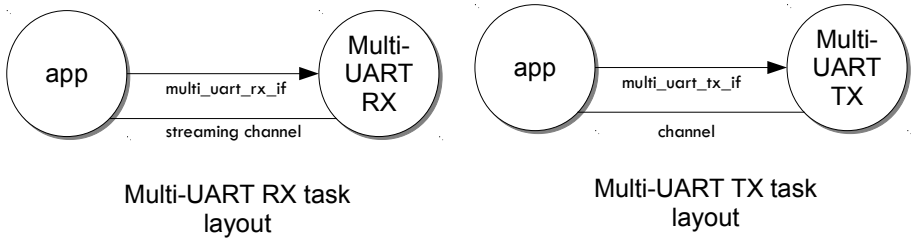

Fig. 9: Multi-UART task diagram

For example, the following code instantiates a multi-UART RX and multi-UART TX component and connects to them:

```
in  buffered port:32 p_uart_rx = XS1_PORT_8A;
out buffered port:8 p_uart_tx  = XS1_PORT_8B;
in  port p_uart_clk            = XS1_PORT_1F;

clock clk_uart = XS1_CLKBLK_4;

int main(void)
{
  interface multi_uart_rx_if i_rx;
  streaming chan c_rx;
  chan c_tx;
  interface multi_uart_tx_if i_tx;

  // Set the rx and tx lines to be clocked off the clk_uart clock block
  configure_in_port(p_uart_rx, clk_uart);
  configure_out_port(p_uart_tx, clk_uart, 0);

  // Configure an external clock for the clk_uart clock block
  configure_clock_src(clk_uart, p_uart_clk);
  start_clock(clk_uart);

  // Start the rx/tx tasks and the application task
  par {
    multi_uart_rx(c_rx, i_rx, p_uart_rx, 8, 1843200, 115200, UART_PARITY_NONE, 8, 1);
    multi_uart_tx(c_tx, i_tx, p_uart_tx, 8, 1843200, 115200, UART_PARITY_NONE, 8, 1);
   app(c_rx, i_rx, c_tx, i_tx);
  }
}
```

The application communicates with all the UARTs via the single multi-UART interfaces e.g.:

```
void loopback(streaming chanend c_rx, client multi_uart_rx_if i_rx,
              chanend c_tx, client multi_uart_tx_if i_tx)
{
  size_t uart_num;

  // Configure each task with a chanend
  i_rx.init(c_rx);
  i_tx.init(c_tx);

  while (1) {
    select {
    case multi_uart_data_ready(c_rx, uart_num):
      uint8_t data;
      if (i_rx.read(uart_num, data) == UART_RX_VALID_DATA) {
        if (i_tx.is_slot_free(uart_num)) {
          i_tx.write(uart_num, data);
        }
        else {
          debug_printf("Warning: TX buffer overflow on channel %d\n",
                       uart_num);
        }
      }
      break;
    }
  }
}
```

Note that the `init` function on the interface must be called once before any use of the interface.

3.4.1 Configuring clocks for multi-UARTs

The ports used for the multi-UART components need to have their clocks configured. For example, the following code configures the multi-UART RX port to run of a clock that is sourced by an incoming port:

```
// Set the rx line to be clocked off the clk_uart clock block
configure_in_port(p_uart_rx, clk_uart);

// Configure an external clock for the clk_uart clock block
configure_clock_src(clk_uart, p_uart_clk);
start_clock(clk_uart);
```

For more information on configuring ports, please refer to the *XMOS Programming Guide* for more details.

The multi-UART components take an argument which is the speed of the underlying clock. This way the component can attain the correct BAUD rate.

The multi-UART RX component must be clocked of a rate which is a multiple of the BAUD rates required.

If a port is not explicitly configured, then it will be clocked of the reference 100Mhz clock of the xcore. The TX component can also work with this clock rate.

3.4.2 Runtime configuration of the Multi-UARTs

The re-configuration of a one of the UARTS in the multi-UART is done via the main `multi_uart_tx_if` or `multi_uart_rx_if`. In both cases, the user must call the `pause` function of the interface, then a reconfiguration function and then the `restart` function e.g.:

```
void app(streaming chanend c_rx, client multi_uart_rx_if i_rx)
  // ...
  i_rx.pause();
  // Set UART number 2 to baud rate 9600
  i_rx.set_baud_rate(2, 9600);
  i_rx.restart();
  // ...
```

4 Examples

Various example application are provided alongside the `lib_uart` which demonstrates the use of the different UART components. These examples can be found in the `examples` directory of the library. All examples provided run on XK-EVK-XU316 board.

4.1 Basic and Streaming UART examples

The basic and streaming UART examples demonstrate the use of the API to loopback data between the UART Tx and Rx components. The examples are designed to be run on a single tile with the UART connection between the *XS1_PORT_1J* and *XS1_PORT_1M* ports (shared with *WIFI MOSI* and *WIFI MISO* on *XK-EVK-XU316*). So make sure to connect these pins with a jumper wire for the example to work.

4.2 Multi-UART example

The multi-UART example demonstrates the use of the multi-UART API to loopback data between multi-UART Tx and Rx components . This example requires two 8-bit ports and a shared clock. The ports chosen are *XS1_PORT_8B* on tile 0 (*X0D14 - X0D21* in the top left header) and *XS1_PORT_8A* on tile 1 (*X1D02 - X1D08* in the bottom left header and *CODEC RST_N* which is *X1D09*). The application will generate a PLL clock on *MCLK* (*X1D11*) which needs to be shared with tile 0 *XS1_PORT_1A* (*X0D00*) port. Make sure to connect 8-bit ports and the share the clock for the example to work.

4.3 Running the examples

This section will describe how to build and run the example applications provided with the `lib_uart` library. The application chosen for this section is the `app_uart_demo` which demonstrates the use of the standard UART API. For other examples, the process is similar, but the application/folder name will change.

4.3.1 Building

The following section assumes that the XMOS XTC tools has been downloaded and installed (see *README* for required version).

Installation instructions can be found here. Particular attention should be paid to the section Installation of required third-party tools.

The application uses the *XMOS* build and dependency system, xcommon-cmake. *xcommon-cmake* is bundled with the *XMOS* XTC tools.

To configure the build, run the following from an XTC command prompt:

```
cd examples
cd app_uart_demo
cmake -G "Unix Makefiles" -B build
```

Any missing dependencies will be downloaded by the build system at this configure step.

Finally, the application binaries can be built using `xmake`:

```
xmake -j -C build
```

4.3.2 Running the application

To run the application return to the `/examples/app_uart_demo` directory and run the following command:

```
xrun --xscope bin/app_uart_demo.xe
```

As application runs and loopbacks data between the UART Tx and Rx components, it will print the received data to the console.

5 UART APIs

5.1 Standard UART API

5.1.1 UART configuration interface

group **`Uart_config_if`**

UART configuration interface.

This interface enables dynamic reconfiguration of a UART. It is used by several UART components to provide a method of configuration.

Functions

void **`set_baud_rate`**(unsigned baud_rate)

Set the baud rate of a UART.

void **`set_parity`**(enum *uart_parity_t* parity)

Set the parity of a UART.

void **`set_stop_bits`**(unsigned stop_bits)

Set number of stop bits used by a UART.

void **`set_bits_per_byte`**(unsigned bits_per_byte)

Set number of bits per byte used by a UART (must be in the range [1-8])

enum **`uart_parity_t`**

Type representing the parity of a UART

Values:

enumerator **`UART_PARITY_EVEN`**

Even parity.

enumerator **`UART_PARITY_ODD`**

Odd parity.

enumerator **UART_PARITY_NONE**

No parity.

5.1.2 UART receiver component

void **uart_rx**(
SERVER_INTERFACE(uart_rx_if, i_data), SERVER_NULLABLE_INTERFACE(uart_config_if, i_config), const_static_unsigned buffer_size, unsigned baud, enum *uart_parity_t* parity, unsigned bits_per_byte, unsigned stop_bits, CLIENT_INTERFACE(input_gpio_if, p_rxd),
)

UART RX.

This function runs a uart receiver. Bytes received by the this task are buffered. When the buffer is full further incoming bytes of data will be dropped. The function never returns and will run indefinitely.

Parameters

- **i_data** – the interface connection allowing clients to receive data
- **i_config** – the interface connection allowing clients to reconfigure the UART
- **buffer_size** – the size of the buffer
- **baud** – the initial baud rate
- **parity** – the initial parity setting
- **bits_per_byte** – the initial number of bits per byte (must be in the range [1-8])
- **stop_bits** – the initial number of stop bits
- **p_rxd** – the gpio interface to input data on

5.1.3 UART receive interface

group **Uart_rx_if**

UART RX interface.

This interface provides clients access to buffer uart receive functionality.

Functions

uint8_t **read**(void)

Get a byte from the receive buffer.

This function should be called after receiving a data_ready() notification. If these is no data in the buffer (for example, this function is called before receiving a notification) then the return value is undefined.

void **data_ready**(void)

Notification that data is in the receive buffer.

This notification function can be selected on by the client and will event when the is data in the receive buffer. After this notification the client should call the read() function.

int **has_data**()

Returns whether there is data in the buffer.

inline uint8_t **wait_for_data_and_read**(CLIENT_INTERFACE(uart_rx_if, i))

Get a byte from the receive buffer.

This function will wait until there is data in the receive buffer of the uart and then fetch that data. On getting the data, it will clear the notification flag on the interface.

5.1.4 UART transmitter components

void **uart_tx**(
SERVER_INTERFACE(uart_tx_if, i_data), SERVER_NULLABLE_INTERFACE(uart_config_if, i_config), unsigned baud, *uart_parity_t* parity, unsigned bits_per_byte, unsigned stop_bits, CLIENT_INTERFACE(output_gpio_if, p_txd),
)

UART transmitter.

This function implements an unbuffered UART transmitter.

Parameters

- **i_data** – interface enabling client to send data
- **i_config** – interface enabling client to configure the UART
- **baud** – the initial baud rate
- **parity** – the initial parity setting
- **bits_per_byte** – the initial number of bits per byte (must be in the range [1-8])
- **stop_bits** – the initial number of stop bits
- **p_txd** – the gpio interface to output data on

void **uart_tx_buffered**(
SERVER_INTERFACE(uart_tx_buffered_if, i_data), SERVER_NULLABLE_INTERFACE(uart_config_if, i_config), const_static_unsigned buffer_size, unsigned baud, *uart_parity_t* parity, unsigned bits_per_byte, unsigned stop_bits, CLIENT_INTERFACE(output_gpio_if, p_txd),
)

UART transmitter (buffered).

This function implements a UART transmitter. Data sent to the task will be placed in a buffer and sent at the rate of the UART.

Parameters

- **i_data** – interface enabling client to send data
- **i_config** – interface enabling client to configure the UART
- **buffer_size** – the size of the transmit buffer in bytes
- **baud** – the initial baud rate
- **parity** – the initial parity setting
- **bits_per_byte** – the initial number of bits per byte (must be in the range [1-8])
- **stop_bits** – the initial number of stop bits
- **p_txd** – the gpio interface to output data on

5.1.5 UART transmit interface

group **Uart_tx_if**

UART transmit interface.

This interface provides functions for transmitting data on an unbuffered UART.

Functions

void **write**(uint8_t data)

Write a byte to a UART.

This function writes a byte of data to a UART. It will output immediately and block until the data is output.

Write a byte to a UART.

This function writes a byte of data to a UART. It will place the data in the output buffer queue to write and then return. If the buffer is full then the data is discarded.

Parameters

- **data** – The data to write.
- **data** – The data to write.

Returns

Zero if the write was successful. If the buffer was full then the function will return 1 and the data is discarded.

5.1.6 UART transmit interface (buffered)

group **Uart_tx_buffered_if**

UART transmit interface (buffered).

This interface contains functions to write to a buffered UART and manage the buffering.

Functions

void **ready_to_transmit**(void)

Ready to transmit notification.

This notification will occur when the UART is ready to transmit (either intially or after a write() call when there is space in the buffer).

size_t **get_available_buffer_size**(void)

Get avaiable buffer size.

This function returns the number of bytes remaining in the buffer that can be filled by write() calls.

5.2 Fast/Streaming API

5.2.1 Streaming receiver

void **uart_rx_streaming**(in_port_t p, streaming_chanend_t c, int ticks_per_bit)

Fast/Streaming UART RX.

This function implements a fast UART. The UART configuration is fixed to a single start bit, 8 bits per byte, and a single stop bit. On a 62.5 MIPS thread this function should be able to keep up with a 10 MBit UART sustained (provided that the streaming channel can keep up with it too).

This function does not return.

Parameters

- **p** – input port, 1 bit port on which data comes in.
- **c** – output streaming channel to connect to the application.
- **ticks_per_bit** – number of clock ticks between bits. This number depends on the clock that is attached to port p. If it is the 100 Mhz reference clock then this value should be at least 10.

void **uart_rx_streaming_read_byte**(
streaming_chanend_t c, REFERENCE_PARAM(uint8_t, data),
)

Receive a byte from a streaming UART receiver.

This function receives a byte from the fast/streaming UART component. It is "select handler" so can be used within a select e.g.

```
uint8_t byte;
size_t index;
select {
  case uart_rx_streaming_receive_byte(c, byte):
       // use sample and index here...
       ...
       break;
...
```

The case in this select will fire when the UART component has data ready.

Parameters

- **c** – chanend connected to the streaming UART receiver component
- **data** – This reference parameter gets set with the incoming data

5.2.2 Streaming transmitter

void **uart_tx_streaming**(out_port_t p, streaming_chanend_t c, int ticks_per_bit)

Fast/Streaming UART TX.

This function implements a fast UART transmitter. It needs an unbuffered 1-bit port, a streaming channel end, and a number of port-clocks to wait between bits. It receives a start bit, 8 bits, and a stop bit, and transmits the 8 bits over the streaming channel end as a single token. On a 62.5 MIPS thread this function should be able to keep up with a 10 MBit UART sustained (provided that the streaming channel can keep up with it too).

This function does not return.

Parameters

- **p** – input port, 1 bit port on which data comes in.
- **c** – output streaming channel to connect to the application.
- **ticks_per_bit** – number of clock ticks between bits. This number depends on the clock that is attached to port p. If it is the 100 Mhz reference clock then this value should be at least 10.

void **uart_tx_streaming_write_byte**(streaming_chanend_t c, uint8_t data)

Write a byte to a streaming UART transmitter.

This function writes a

Parameters

- **c** – chanend connected to the streaming UART Tx component
- **data** – The data to send.

5.3 Half-Duplex API

5.3.1 Half-duplex component

void **uart_half_duplex**(
SERVER_INTERFACE(uart_tx_buffered_if, i_tx), SERVER_INTERFACE(uart_rx_if, i_rx), SERVER_INTERFACE(uart_control_if, i_control), SERVER_NULLABLE_INTERFACE(uart_config_if, i_config), const_static_unsigned tx_buf_length, const_static_unsigned rx_buf_length, unsigned baud, *uart_parity_t* parity, unsigned bits_per_byte, unsigned stop_bits, port p_uart,
)

Half duplex UART.

This function implements a UART that can either transmit or receive on the same wire. The application explicitly control whether the component is in transmit or receive mode.

Parameters

- **i_tx** – interface for transmitting data
- **i_rx** – interface for receiving data
- **i_control** – interface for controlling the direction of the UART
- **i_config** – interface for configuring the UART
- **tx_buf_length** – the size of the transmit buffer (in bytes)
- **rx_buf_length** – the size of the receive buffer (in bytes)
- **baud** – baud rate
- **parity** – the parity of the UART
- **bits_per_byte** – bits per byte (must be in the range [1-8])
- **stop_bits** – The number of stop bits
- **p_uart** – the 1-bit port to send/recieve the UART signals.

5.3.2 Half-duplex control interface

enum **uart_half_duplex_mode_t**

Type representing the mode (direction) of a uart.

Values:

enumerator **UART_RX_MODE**

Uart is in receive mode.

enumerator **UART_TX_MODE**

Uart is in transmit mode.

group **Uart_control_if**

Interface to control the mode of a half-duplex UART

Functions

void **set_mode**(*uart_half_duplex_mode_t* mode)

Set the mode of the UART.

This function can be used to control whether the UART is in send or receive mode.

5.4 Multi-UART API

5.4.1 Multi-UART receiver

void **multi_uart_rx**(
streaming_chanend_t c, SERVER_INTERFACE(multi_uart_rx_if, i), in_buffered_port_32_t p, clock clk, size_t num_uarts, unsigned clock_rate_hz, unsigned baud, enum *uart_parity_t* parity, unsigned bits_per_byte, unsigned stop_bits,
)

Multi-UART receiver.

This function implements multiple UART receivers on a multi-bit port. The UARTS all have the same baud rate. The parity, bits per byte and number of stop bits is the same for all UARTs and cannot be changed dynamically.

Parameters

- **c** – a chanend used internally for high speed communication
- **i** – the interface for getting data from the task
- **p** – the multibit port
- **clk** – a clock block for the component to use. This needs to be set to run of the reference clock (the default state for clock blocks)
- **num_uarts** – the number of uarts to run (must be less than or equal to the width of **p**)
- **clock_rate_hz** – the clock rate in Hz
- **baud** – baud rate
- **parity** – the parity of the UART
- **bits_per_byte** – bits per byte (must be in the range [1-8])
- **stop_bits** – number of stop bits

5.4.2 Multi-UART receive interface

enum **multi_uart_read_result_t**

Values:

enumerator **UART_RX_VALID_DATA**

Data received is valid.

enumerator **UART_RX_INVALID_DATA**

Data received is not valid.

group **Multi_uart_rx_if**

Multi-UART receive interface

Functions

void **init**(streaming_chanend_t c)

Initialize the multi-UART RX component.

Parameters

- **c** – The chanend connected to the multi-UART RX task

enum *multi_uart_read_result_t* **read**(
size_t index, REFERENCE_PARAM(uint8_t, data),
)

Read a byte for the next UART with ready data.

This function will read out a byte from the next UART with data available. If several UARTS have data available then the data is read out in a round-robin fashion.

Parameters

- **index** – This index of the UART to read from
- **data** – The data byte read

Returns

An enum type that indicates if the data is valid

void **pause**(void)

Pause the multi-UART RX component for reconfiguration.

This call will stop the mulit-UART component so that the UARTs can be reconfigured.

void **restart**(void)

Restart the multi-UART RX component after reconfiguration.

This call will restart the multi-UART component.

void **set_baud_rate**(size_t index, unsigned baud_rate)

Set the baud rate of a UART.

This call will set the baud rate of one of the UARTs. The rate must be a divisor of the clock rate of the underlying clock used for the component.

Set the baud rate of a UART.

This call will set the baud rate of one of the UARTs. The rate must be a divisor of the clock rate of the underlying clock used for the component.

Parameters

- **index** – The index of the UART to configure
- **baud_rate** – The required baud rate
- **index** – The index of the UART to configure.
- **baud_rate** – The required baud rate

void **set_parity**(size_t index, enum *uart_parity_t* parity)

Set parity of a UART.

This call will set the parity of one of the UARTs. The rate must be a divisor of the clock rate of the underlying clock used for the component.

Parameters

- **index** – The index of the UART to configure.
- **parity** – The required parity

void **set_stop_bits**(size_t index, unsigned stop_bits)

Set the number of stop bits of a UART.

This call will set the number of stop bits of one of the UARTs.

Parameters

- **index** – The index of the UART
- **stop_bits** – The number of stop bits

void **set_bits_per_byte**(size_t index, unsigned bits_per_byte)

Set the number of bit per byte of a UART.

This call will set the number of stop bits of one of the UARTs.

Parameters

- **index** – The index of the UART
- **bits_per_byte** – The number of bits per byte (must be in the range [1-8])

5.4.3 Multi-UART transmitter

void **multi_uart_tx**(
chanend c, SERVER_INTERFACE(multi_uart_tx_if, i), out_port_t p, size_t num_uarts, unsigned clock_rate_hz, unsigned baud, *uart_parity_t* parity, unsigned bits_per_byte, unsigned stop_bits,
)

Multi-UART transmitter.

This function implements multiple UART transmiiters on a multi-bit port. The UARTS all have the same baud rate. The parity, bits per byte and number of stop bits is the same for all UARTs and cannot be changed dynamically.

Parameters

- **c** – a chanend used internally for high speed communication
- **i** – the interface for sending data to the task
- **p** – the multibit port
- **num_uarts** – the number of uarts to run (must be less than or equal to the width of **p**)
- **clock_rate_hz** – the clock rate in Hz
- **baud** – baud rate
- **parity** – the parity of the UART
- **bits_per_byte** – bits per byte (must be in the range [1-8])
- **stop_bits** – number of stop bits

5.4.4 Multi-UART transmit interface

group **`Multi_uart_tx_if`**

Multi-UART transmit interface

Functions

void **`init`**(chanend c)

Initialize the multi-UART TX component.

Parameters

- **`c`** – The chanend connected to the multi-UART TX task

int **`is_slot_free`**(size_t index)

Check whether transmit slot is free.

This function checks whether the application can write data to a specific UART.

Parameters

- **`index`** – The index of the UART to check

Returns

non-zero if the slot is free (i.e. data can be sent)

void **`write`**(size_t index, uint8_t data)

Write to a UART.

This function writes a byte of data to a UART. This byte will be buffered to send. If the transmit buffer for that UART is not available then the data is ignored (use is_tx_slot_free() to determine availability).

Parameters

- **`index`** – The index of the UART to write to
- **`data`** – The data to write