UART library

A software defined, industry-standard, UART (Universal Asynchronous Receiver/Transmitter) library that allows you to control an UART serial connection via the xCORE GPIO hardware-response ports. This library is controlled via C using the XMOS multicore extensions.

Features

- UART receive and transmit
- Supports speeds up to 10MBit/s
- Half-duplex mode (applicable to RS485)
- Efficient multi-uart mode for implementing multiple connections

Resource Usage

This following table shows typical resource usage in some different configurations. Exact resource usage will depend on the particular use of the library by the application.

Configuration	Pins	Ports	Clocks	Ram	Logical cores
Standard TX	1	1	0	~1.0K	0
Standard TX (buffered)	1	1	0	~1.2K	$\leq$ 1
Standard RX	1	1	0	~1.5K	$\leq$ 1
Fast/streaming TX	1	1	0	~0.2K	1
Fast/streaming RX	1	1	0	~0.2K	1
Multi-UART TX (8 UARTs)	8	1	0	~2.9K	1
Multi-UART RX (8 UARTs)	8	1	0	~3.4K	1
Half Duplex	1	1	0	~1.8K	1

Software version and dependencies

This document pertains to version 3.0.2 of this library. It is known to work on version 14.1.1 of the xTIMEcomposer tools suite, it may work on other versions.

This library depends on the following other libraries:

- lib_logging (>=2.0.0)
- lib_xassert (>=2.0.0)
- lib_gpio (>=1.0.0)

Related application notes

The following application notes use this library:

- AN00158 - How to use the UART lbirary
- AN00159 - How to run large numbers of UARTS
- AN00163 - Using half duplex UARTS over RS485

1 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 Figure 1. The data is driven least significant bit first.

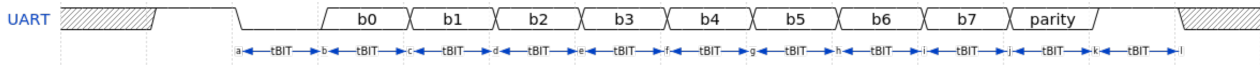

Figure 1: UART data sequence

The start bit, data bits, parity bit and stop bits are all the same length (tBIT in Figure 1). This length is give by the BAUD rate which is the number of bits per second.

1.1 Connecting to the xCORE device

If you are using the general 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).

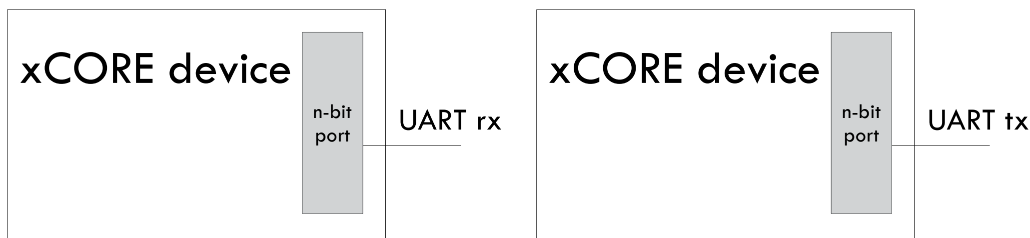

Figure 2: UART Rx and Tx connections

The half duplex UART needs to be connected to a 1-bit port.

Figure 3: UART half duplex connection

Copyright 2016 XMOS Ltd.

The fast/streaming UART also needs to be connect to a 1-bit port for TX or RX.

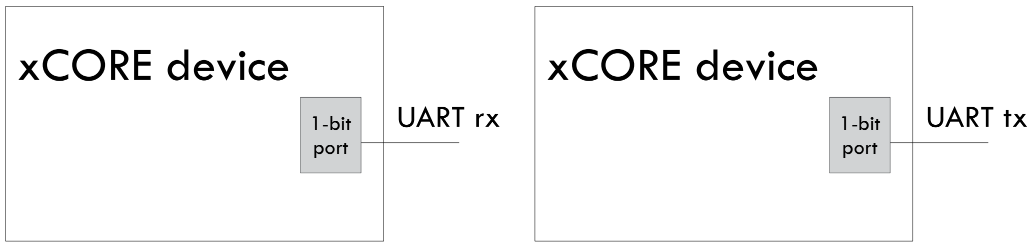

Figure 4: Fast/Streaming UART connections

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.

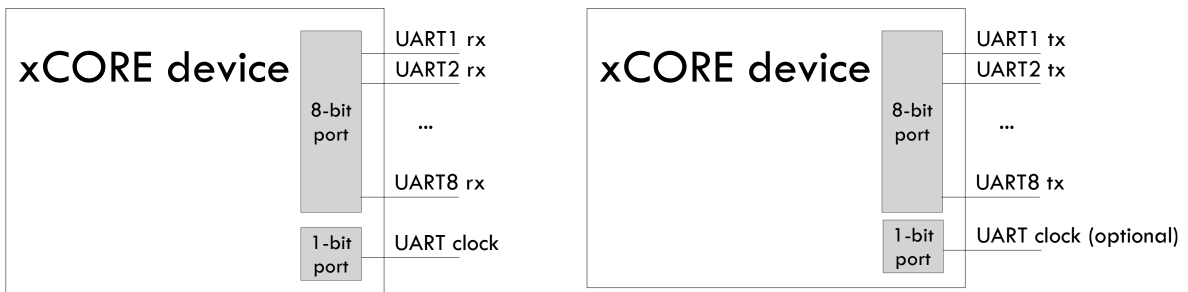

Figure 5: Multi UART connections

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 1843200Khz oscillator is a multiple of 115200 baud (and lower rates also). The maximum allowable incoming signal is 1843200Khz.

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).

Copyright 2016 XMOS Ltd.

2 Usage

The 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 runs 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 (link?)
Multi-UART	The multi-UART components efficiently run several UARTS on the same core using a multibit port.

All the UARTs use the XMOS multicore extensions to C (xC) to perform their operations, see the XMOS Programming Guide (see XM-004440-PC) for more details.

2.1 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). Both components also have an optional configuration interface that lets the application change the speed and properties of the UART at run time.

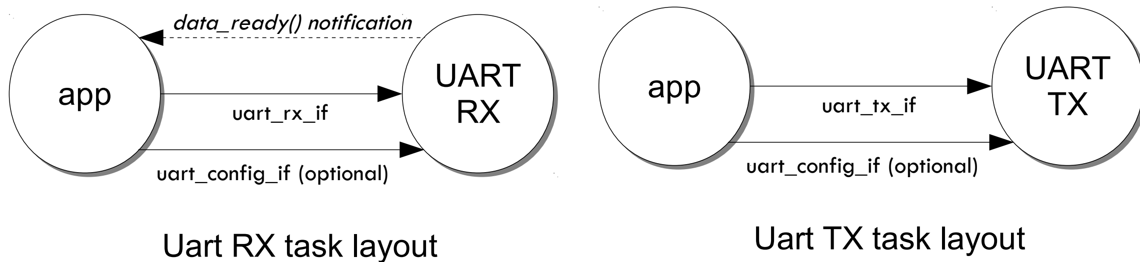

Figure 6: UART task diagram

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.

Copyright 2016 XMOS Ltd.

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;
   }
}
```

2.1.1 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.

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

2.2 Fast/Streaming UART usage

The fast/streaming UART components are instantiated as parallel tasks that run in a par statement. The can connect via a streaming channel.

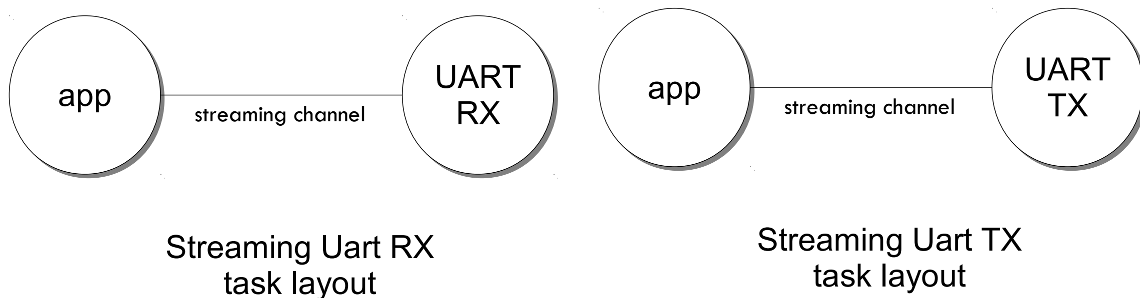

Figure 7: Fast/streaming UART task diagram

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 §4) 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);
   ...
```

Copyright 2016 XMOS Ltd.

2.3 Half-duplex UART usage

The half-duplexcomponents are instantiated as parallel tasks that run in a par statement. The application connects via an 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). The component also has an optional configuration interface that lets the application change the speed and properties of the UART at run time.

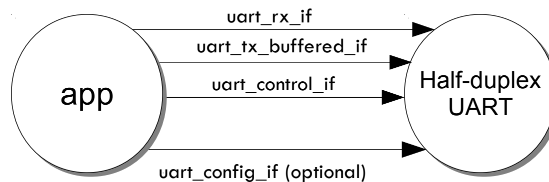

Figure 8: Half-duplex UART task diagram

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_control1;
  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);
   ...
```

Copyright 2016 XMOS Ltd.

2.4 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.

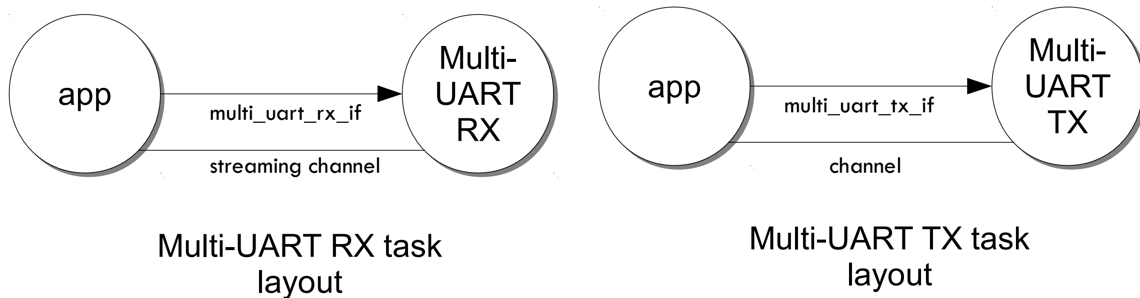

Figure 9: Multi-UART task diagram

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);
  }
}
```

Copyright 2016 XMOS Ltd.

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.

2.4.1 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 (see XM-004440-PC) 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.

2.4.2 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();
  ....
```

Copyright 2016 XMOS Ltd.

3 Standard UART API

3.1 UART configuration interface

Type	uart_config_if
Description	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	

Function	set_baud_rate
Description	Set the baud rate of a UART.
Type	`void set_baud_rate(unsigned baud_rate)`

Function	set_parity
Description	Set the parity of a UART.
Type	`void set_parity(enum uart_parity_t parity)`

Function	set_stop_bits
Description	Set number of stop bits used by a UART.
Type	`void set_stop_bits(unsigned stop_bits)`

Function	set_bits_per_byte
Description	Set number of bits per byte used by a UART.
Type	`void set_bits_per_byte(unsigned bpb)`

Type	uart_parity_t
Description	Type representing the parity of a UART.
Values	UART_PARITY_EVEN Even parity. UART_PARITY_ODD Odd parity. UART_PARITY_NONE No parity.

3.2 UART receiver component

<table>
<tr><td>Function</td><td>uart_rx</td></tr>
<tr><td>Description</td><td>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.</td></tr>
<tr><td>Type</td><td><pre>[[combinable]]
void
uart_rx(server interface uart_rx_if i_data,
   server 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 input_gpio_if p_rxd)</pre></td></tr>
<tr><td>Parameters</td><td><code>i_data</code> the interface connection allowing clients to receive data

<code>i_config</code> the interface connection allowing clients to reconfigure the UART

<code>buffer_size</code>
the size of the buffer

<code>baud</code> the initial baud rate

<code>parity</code> the intiial parity setting

<code>bits_per_byte</code>
the initial number of bits per byte

<code>stop_bits</code> the intiial number of stop bits

<code>p_rxd</code> the gpio interface to input data on</td></tr>
</table>

3.3 UART receive interface

Type	**uart_rx_if**
Description	UART RX interface. This interface provides clients access to buffer uart receive functionality.
Functions	

Function	**read**
Description	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.
Type	`[[clears_notification]]` `uint8_t read(void)`

Function	**data_ready**
Description	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.
Type	`[[notification]]` `slave void data_ready(void)`

Function	**has_data**
Description	Returns whether there is data in the buffer.
Type	`int has_data()`

Function	**wait_for_data_and_read**
Description	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.
Type	`uint8_t wait_for_data_and_read()`

Copyright 2016 XMOS Ltd.

3.4 UART transmitter components

Function	uart_tx
Description	UART transmitter. This function implements an unbuffered UART transmitter.

Type	`[[distributable]]`
	`void`
	`uart_tx(server interface uart_tx_if i_data,`
	`server interface uart_config_if ?i_config,`
	`unsigned baud,`
	`uart_parity_t parity,`
	`unsigned bits_per_byte,`
	`unsigned stop_bits,`
	`client output_gpio_if p_txd)`

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 intiial parity setting. bits_per_byte — the initial number of bits per byte. stop_bits — the intiial number of stop bits. p_txd — the gpio interface to output data on.

<table>
<tr><td>Function</td><td>uart_tx_buffered</td></tr>
<tr><td>Description</td><td>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.</td></tr>
<tr><td>Type</td><td><pre>[[combinable]]
void
uart_tx_buffered(server interface uart_tx_buffered_if i_data,
   server 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 output_gpio_if p_txd)</pre></td></tr>
<tr><td>Parameters</td><td><code>i_data</code> interface enabling client to send data.

<code>i_config</code> interface enabling client to configure the UART.

<code>buffer_size</code>
the size of the transmit buffer in bytes.

<code>baud</code> the initial baud rate.

<code>parity</code> the intiial parity setting.

<code>bits_per_byte</code>
the initial number of bits per byte.

<code>stop_bits</code> the intiial number of stop bits.

<code>p_txd</code> the gpio interface to output data on.</td></tr>
</table>

3.5 UART transmit interface

<table>
<tr><td>Type</td><td>uart_tx_if</td></tr>
<tr><td>Description</td><td>UART transmit interface.
This interface provides functions for transmitting data on an unbuffered UART.</td></tr>
<tr><td>Functions</td><td>
<table>
<tr><td>Function</td><td>write</td></tr>
<tr><td>Description</td><td>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.</td></tr>
<tr><td>Type</td><td><code>void write(uint8_t data)</code></td></tr>
<tr><td>Parameters</td><td><code>data</code> The data to write.</td></tr>
</table>
</td></tr>
</table>

3.6 UART transmit interface (buffered)

Type	uart_tx_buffered_if
Description	UART transmit interface (buffered). This interface contains functions to write to a buffered UART and manage the buffering.
Functions	

Function	write
Description	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.
Type	`[[clears_notification]]` `int write(uint8_t data)`
Parameters	`data` The data to write.
Returns	non-zero if the write was succesfully. If the buffer was full then the function will return zero.

Function	ready_to_transmit
Description	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).
Type	`[[notification]]` `slave void ready_to_transmit(void)`

Function	get_available_buffer_size
Description	Get avaiable buffer size. This function returns the number of bytes remaining in the buffer that can be filled by `write()` calls.
Type	`size_t get_available_buffer_size(void)`

Copyright 2016 XMOS Ltd.

4 Fast/Streaming API

4.1 Streaming receiver

Function	**uart_rx_streaming**
Description	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.
Type	`void uart_rx_streaming(in port p, streaming chanend c, int ticks_per_bit)`
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.

Function	**uart_rx_streaming_read_byte**

Description	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.

Type	`void uart_rx_streaming_read_byte(streaming chanend c, uint8_t &data)`
Parameters	c — chanend connected to the streaming UART receiver component data — This reference parameter gets set with the incoming data

Copyright 2016 XMOS Ltd.

4.2 Streaming transmitter

Function	uart_tx_streaming
Description	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.

Type	`void uart_tx_streaming(out port p,`
	`                       streaming chanend c,`
	`                       int ticks_per_bit)`

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.

Function	uart_tx_streaming_write_byte
Description	Write a byte to a streaming UART transmitter. This function writes a

Type	`void`
	`uart_tx_streaming_write_byte(streaming chanend c,`
	`                             uint8_t data)`

Parameters	`c` chanend connected to the streaming UART Tx component `data` The data to send.

5 Half-Duplex API

5.1 Half-duplex component

<table>
<tr><td>Function</td><td>uart_half_duplex</td></tr>
<tr><td>Description</td><td>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.</td></tr>
<tr><td>Type</td><td><pre>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 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)</pre></td></tr>
<tr><td>Parameters</td><td>
<table>
<tr><td>i_tx</td><td>interface for transmitting data.</td></tr>
<tr><td>i_rx</td><td>interface for receiving data.</td></tr>
<tr><td>i_control</td><td>interface for controlling the direction of the UART.</td></tr>
<tr><td>i_config</td><td>interface for configuring the UART.</td></tr>
<tr><td>tx_buf_length</td><td>the size of the transmit buffer (in bytes).</td></tr>
<tr><td>rx_buf_length</td><td>the size of the receive buffer (in bytes).</td></tr>
<tr><td>baud</td><td>baud rate.</td></tr>
<tr><td>parity</td><td>the parity of the UART.</td></tr>
<tr><td>bits_per_byte</td><td>bits per byte.</td></tr>
<tr><td>stop_bits</td><td>The number of stop bits (0,1 or 2)</td></tr>
<tr><td>p_uart</td><td>the 1-bit port to send/recieve the UART signals.</td></tr>
</table>
</td></tr>
</table>

Copyright 2016 XMOS Ltd.

5.2 Half-duplex control interface

Type	**uart_half_duplex_mode_t**
Description	Type representing the mode (direction) of a uart.
Values	UART_RX_MODE Uart is in receive mode. UART_TX_MODE Uart is in transmit mode.

Type	**uart_control_if**
Description	Interface to control the mode of a half-duplex UART.
Functions	

Function	**set_mode**
Description	Set the mode of the UART. This function can be used to control whether the UART is in send or receive mode.
Type	`void` `set_mode(uart_half_duplex_mode_t mode)`

6 Multi-UART API

6.1 Multi-UART receiver

<table>
<tr><td>Function</td><td>multi_uart_rx</td></tr>
<tr><td>Description</td><td>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.</td></tr>
<tr><td>Type</td><td><pre>void multi_uart_rx(streaming chanend c,
                   server interface multi_uart_rx_if i,
                   in buffered port:32 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)</pre></td></tr>
<tr><td>Parameters</td><td>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)

clock_rate_hz — the clock rate in Hz

baud — baud rate.

parity — the parity of the UART.

bits_per_byte — bits per byte.

stop_bits — number of stop bits.</td></tr>
</table>

Copyright 2016 XMOS Ltd.

6.2 Multi-UART receive interface

Type	**multi_uart_read_result_t**
Description	
Values	UART_RX_VALID_DATA Data received is valid. UART_RX_INVALID_DATA Data received is not valid.

Type	**multi_uart_rx_if**
Description	Multi-UART receive interface.
Functions	

Function	**init**
Description	Initialize the multi-UART RX component.
Type	`void init(streaming chanend c)`
Parameters	`c` The chanend connected to the multi-UART RX task

Function	**read**
Description	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.
Type	`enum multi_uart_read_result_t read(size_t index, uint8_t &data)`
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

Continued on next page

Type	multi_uart_rx_if (continued)

Function	pause
Description	Pause the multi-UART RX component for reconfiguration. This call will stop the mulit-UART component so that the UARTs can be reconfigured.
Type	`void pause()`

Function	restart
Description	Restart the multi-UART RX component after reconfiguration. This call will restart the multi-UART component.
Type	`void restart()`

Function	set_baud_rate
Description	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.
Type	`void set_baud_rate(size_t index, unsigned baud_rate)`
Parameters	`index` The index of the UART to configure. `baud_rate` The required baud rate

Function	set_parity
Description	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.
Type	`void set_parity(size_t index, enum uart_parity_t parity)`
Parameters	`index` The index of the UART to configure. `parity` The required parity

Continued on next page

Type	multi_uart_rx_if (continued)

Function	set_stop_bits
Description	Set the number of stop bits of a UART. This call will set the number of stop bits of one of the UARTs.
Type	`void set_stop_bits(size_t index, unsigned stop_bits)`
Parameters	`index` The index of the UART `stop_bits` The number of stop bits (0,1 or 2)

Function	set_bits_per_byte
Description	Set the number of bit per byte of a UART. This call will set the number of stop bits of one of the UARTs.
Type	`void set_bits_per_byte(size_t index, unsigned bpb)`
Parameters	`index` The index of the UART bpb The number of bits per byte (5,6,7 or 8)

6.3 Multi-UART transmitter

Function	multi_uart_tx
Description	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.

Type	`void multi_uart_tx(chanend c,`
	`server interface multi_uart_tx_if i,`
	`out port p,`
	`size_t num_uarts,`
	`unsigned clock_rate_hz,`
	`unsigned baud,`
	`uart_parity_t parity,`
	`unsigned bits_per_byte,`
	`unsigned stop_bits)`

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) `clock_rate_hz` the clock rate in Hz `baud` baud rate. `parity` the parity of the UART. `bits_per_byte` bits per byte. `stop_bits` number of stop bits.

Copyright 2016 XMOS Ltd.

6.4 Multi-UART transmit interface

Type	**multi_uart_tx_if**
Description	Multi-UART transmit interface.
Functions	

Function	**init**	
Description	Initialize the multi-UART TX component.	
Type	`void init(chanend c)`	
Parameters	`c`	The chanend connected to the multi-UART TX task

Function	**is_slot_free**	
Description	Check whether transmit slot is free. This function checks whether the application can write data to a specific UART.	
Type	`int is_slot_free(size_t index)`	
Parameters	`index`	The index of the UART to check.
Returns	non-zero if the slot is free (i.e. data can be sent).	

Function	**write**	
Description	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).	
Type	`void write(size_t index, uint8_t data)`	
Parameters	`index`	The index of the UART to write to.
	`data`	The data to write.

Continued on next page

Copyright 2016 XMOS Ltd.

Type	multi_uart_tx_if (continued)

Function	pause
Description	Pause the multi-UART RX component for reconfiguration. This call will stop the mulit-UART component so that the UARTs can be reconfigured.
Type	`void pause()`

Function	restart
Description	Restart the multi-UART RX component after reconfiguration. This call will restart the multi-UART component.
Type	`void restart()`

Function	set_baud_rate
Description	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.
Type	`void set_baud_rate(size_t index, unsigned baud_rate)`
Parameters	`index` The index of the UART to configure. `baud_rate` The required baud rate

Function	set_parity
Description	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.
Type	`void set_parity(size_t index, enum uart_parity_t parity)`
Parameters	`index` The index of the UART to configure. `parity` The required parity

Continued on next page

Type	multi_uart_tx_if (continued)

Function	set_stop_bits
Description	Set the number of stop bits of a UART. This call will set the number of stop bits of one of the UARTs.
Type	`void set_stop_bits(size_t index, unsigned stop_bits)`
Parameters	`index` The index of the UART `stop_bits` The number of stop bits (0,1 or 2)

Function	set_bits_per_byte
Description	Set the number of bit per byte of a UART. This call will set the number of stop bits of one of the UARTs.
Type	`void set_bits_per_byte(size_t index, unsigned bpb)`
Parameters	`index` The index of the UART `bpb` The number of bits per byte (5,6,7 or 8)

Copyright 2016 XMOS Ltd.

APPENDIX A - Known Issues

No known issues.

Copyright 2016 XMOS Ltd.

APPENDIX B - UART Library Change Log

B.1 3.0.2

- Update to source code license and copyright

B.2 3.0.1

- Update fast rx and tx to match API prototypes & fix port directions
- Fixed order of ports in api calls from example program

B.3 3.0.0

- Restructued version
- Changes to dependencies:
  - lib_logging: Added dependency 2.0.0
  - lib_xassert: Added dependency 2.0.0
  - lib_gpio: Added dependency 1.0.0

B.4 2.3.2

- Increment version for XPD release. Several minor docs bugs fixed.

B.5 2.3.1

- Tidied up uart_fast and targetted demo at L16 sliceKIT

B.6 3.0.0

- Major change to generic UART tx/rx components to use new xC features with different api.

B.7 2.3.0

- Added RS485 component and apps

B.8 2.2.0

- Updated documents for xSOFTip requirements
- Added metainfo and XPD items

B.9 2.1.0

- Documentation Updates

Copyright © 2016, All Rights Reserved.

Xmos Ltd. is the owner or licensee of this design, code, or Information (collectively, the "Information") and is providing it to you "AS IS" with no warranty of any kind, express or implied and shall have no liability in relation to its use. Xmos Ltd. makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.