Stand Alone Uart Components

REV A

Publication Date: 2012/10/16

XMOS

Table of Contents

1 Overview

IN THIS CHAPTER

UART is one of the most basic asynchronous communication protocols. Also known as RS-232, it transmits bits serially at a mutually agreed speed without providing a clock. The speed is known as the *baud* rate, and is typically something like 9600 baud, 115200 baud, or 10 Mbaud.

The most important characteristics are the following:

- The baud rate.
- The number of start bits, typically 1, sometimes 1.5
- The number of data bits, typically 7 or 8
- Whether a parity bit is present, and if so whether it is even parity or odd parity
- The number of stop bits, typically 1, sometimes 2.

1.1 Uart Modules

Four uart modules are covered by this document, as shown below.

Component	module	Data rate	Memory	Parity	Bits Per Byte	Stop Bits
Generic UART	module_uart_rx	115.2K	~1K	Even/Odd/None	5/6/7/8	1 or 2
	module_uart_tx	115.2K	~1K	Even/Odd/None	5/6/7/8	1 or 2
Simple UART	module_uart_fast_rx	10 Mbaud	0.25K	None	8	1
	module_uart_fast_tx	10 Mbaud	0.25K	None	8	1

1.2 Generic UART

This module is completely parameterisable at run time and will require a logical core for each of RX and TX. Unlike the simple uart below, it can operate at the standard UART baud rates.

1.3 Simple Uart

This module is a much simpler implementation of a UART that will require a whole logical core for RX and deliver up to 10 Mbaud. TX could be called as a function, and hence share a logical core with other functionality although this will affect the TX baud rate achieved,and this usage is not shown.

It is fixed to 8 bits, a single start bit, no parity, and a single stop bit. All of those parameters could be changed by altering the source code.

The baud rate is parameterisable, but has to be a whole division of 100 MHz. This may make it unsuitable for some applications requiring a very particular baud rate.

Note that the code is easy to understand; it comprises the example from the XC programming manual.

1.4 Other Uarts

For an implementation of multi-uart suitable for applications where more than one uart is required, the module_multi_uart component may be a better choice, which offers up t 8 uarts in two logical cores. In general, designs requring a single basic uarts are often most optimally constructed by incorporating the TX functionality inline with other application functions, and having a separate logical core for RX. A fully optimised design incorporating uarts may therefore merit some modifications of these components.

2 Generic UART

IN THIS CHAPTER

This UART has two modules, one for RX and one for TX each of which uses one logical core and is connected via channel to another logical core using the UART Client API .

The logical core for the UART TX component receives data from the client via a buffer (configurable between 1 and 64 bytes). A full buffer will block the client.

The logical core for the UART RX component receives data from external into the RX buffer (configurable between 1 and 64 bytes) which is read by the client using channel. An empty RX buffer will block the client.

2.1 Resource Requirements

A single generic uart consisting of module_uart_rx and module_uart_tx will consumer the following resources:

Resource	Usage
Code Memory	2110
Ports	2 x 1b
ChannelEnds	2

2.2 Evaluation Platforms

2.2.1 Recommended Hardware

This module may be evaluated using the Slicekit Modular Development Platform, available from digikey. Required board SKUs are:

- XP-SKC-L2 (Slicekit L2 Core Board) plus XA-SK-GPIO plus XA-SK-ATAG2 (Slicekit XTAG adaptor) plus XTAG2 (debug adaptor), OR
- XK-SK-L2-ST (Slicekit Starter Kit, containing all of the above).

2.2.2 Demonstration Application

Example usage of this module can be found within the XSoftIP suite as follows:

- Package: sw_gpio_examples
- Application: app_slicekit_com_demo
- Package: sc_uart
- Application: app_uart_back2back

2.3 API and Programming Guide

2.3.1 Key Files

File	Description
module_uart_tx/src/uart_tx.h	Client API header file for Generic UART TX
module_uart_tx/src/uart_tx.xc	Client API implementation for Generic UART TX
module_uart_tx/src/uart_tx_impl.h	UART TX Server Header File
module_uart_tx/src/uart_tx_impl.xc	UART TX Server implementation
module_uart_rx/src/uart_rx.h	Client API header file for Generic UART RX
module_uart_rx/src/uart_rx.xc	Client API implementation for Generic UART RX
module_uart_rx/src/uart_rx_impl.h	UART RX Server Header File
module_uart_rx/src/uart_rx_impl.xc	UART RX Server implementation

2.3.2 TX API

The UART TX component is started using the uart_tx() function, which causes the TX server (called "_impl" in the code) to run in a while(1) loop until it receives and instruction to shut down via the channel from the client. Additionally a set of client functions are provided to transmit a byte and to alter the baud rate, parity, bits per byte and stop bit settings. Any such action except transmitting a byte causes the TX server to terminate and then be restarted with the new settings. Any data in the TX buffer is preserved and then sent with the new settings.

```
void uart_tx(out port txd,
             unsigned char buffer[],
             unsigned buffer_size,
             unsigned baud_rate,
             unsigned bits,
             enum uart_tx_parity parity,
             unsigned stop_bits,
             chanend c)
```

UART TX server function.

This function never returns and the arguments specify the initial configuration of parity, bits per byte etc. These are checked and then the inner UART TX

loop (`uart_tx_impl()` is started and runs in a while(1) loop until it receives and instruction to shut down via the channel from the client. Additionally a set of client functions are provided to transmit a byte and to alter the baud rate, parity, bits per byte and stop bit settings. Note that the buffer size can only be changed when this function is first called, it cannot be changed thereafter.

The client provides data to the server using the send byte function below.

Arguments are as follows:

This function has the following parameters:

`txd`	Transmitted data signal. Must be a 1-bit port.
`buffer`	Array for buffering bytes before they are transmitted.
`buffer_size`	Size of the array.
`baud_rate`	Initial baud rate.
`bits`	Initial bits per character. Must be less than 8.
`parity`	Type of parity to initially use. No parity bit is transmitted if is passed.
`stop_bits`	Initial number of stop bits to transmit.
`c`	Chanend to receive requests on, where a request is either a byte to transmit or a configuration change.

`void uart_tx_send_byte(chanend c, unsigned char byte)`

Adds a byte into the UART transmit buffer.

If the the buffer is full then this function blocks until there is room in the buffer.

This function has the following parameters:

`c`	Other end of the channel passed to the uart_tx() function.
`byte`	Byte to transmit.

`void uart_tx_set_baud_rate(chanend c, unsigned baud_rate)`

Set the baud rate of the of the UART TX server.

The change of configuration takes effect after all remaining data in transmit buffer is sent, and the function blocks until the change of configuration takes effect.

This function has the following parameters:

`c`	Other end of the channel passed to the uart_tx() function.
`baud_rate`	The baud rate setting.

`void uart_tx_set_parity(chanend c, enum uart_tx_parity parity)`

Set the parity of the of the UART TX stop bit.

The change of configuration takes effect after all remaining data in transmit buffer is sent, and the function blocks until the change of configuration takes effect.

This function has the following parameters:

`c` Other end of the channel passed to the uart_tx() function.

`parity` The parity setting. Note parity is an enum.

`void uart_tx_set_stop_bits(chanend c, unsigned stop_bits)`

Set the number of stop bits used by the UART TX server.

The change of configuration takes effect after all remaining data in transmit buffer is sent, and the function blocks until the change of configuration takes effect.

This function has the following parameters:

`c` Other end of the channel passed to the uart_tx() function.

`stop_bits` The stop bits or bits-per-byte setting. Note parity is an enum.

`void uart_tx_set_bits_per_byte(chanend c, unsigned bits)`

Set the bits per byte rate of the of the UART TX server.

The change of configuration takes effect after all remaining data in transmit buffer is sent, and the function blocks until the change of configuration takes effect.

This function has the following parameters:

`c` Other end of the channel passed to the uart_tx() function.

`bits` The bits or bits-per-byte setting.

2.3.3 RX API

The UART RX component is started using the uart_tx() function, which causes the TX server (called "_impl" in the code) to run in a while(1) loop until it receives and instruction to shut down via the channel from the client. Additionally a set of client functions are provided to fetch a byte from teh recieve buffer and to alter the baud rate, parity, bits per byte and stop bit settings. Any such action except transmitting a byte causes the RX server to terminate, the receive buffer emptied, and then restarted immediately with the new settings.

```
void uart_rx(in buffered port:1 rxd,
             unsigned char buffer[],
             unsigned buffer_size,
             unsigned baud_rate,
             unsigned bits,
             enum uart_rx_parity parity,
```

```
            unsigned stop_bits,
            chanend c)
```

UART RX server function.

The client must call the `uart_rx_init()` function to initialize the server before calling any other client functions. Bytes received by the server are buffered in the array. When the buffer is full further incoming bytes of data will be dropped.

This function never returns, and will start the server RX inner loop which runs in a while(1) loop until it is stopped via a command over channelend, following whcih the inner loop is automatyically restarted.

The functions below are provided to change the UART RX parameters, parity, bits-per-byte, stop bits and baud rate, and all of these cause the RX inner loop to terminate while the new settings are applied. Bytes received during this time would be dropped, and any receive buffer contents are discarded when the settings are changed. Note that the buffer size can only be changed when this function is first called, it cannot be changed thereafter.

The client uses uart_rx_get_byte() to get bytes from the receive buffer.

```
unsigned char uart_rx_get_byte(chanend c, uart_rx_client_state &state)
```

Get a byte from the receive buffer.

This function blocks until there is a byte available.

```
void uart_rx_get_byte_byref(chanend c,
                            uart_rx_client_state &state,
                            unsigned char &byte)

void uart_rx_set_baud_rate(chanend c,
                           uart_rx_client_state &state,
                           unsigned baud_rate)
```

Set the baud rate of the UART RX server.

The change of configuration takes place immediately.

```
void uart_rx_set_parity(chanend c,
                        uart_rx_client_state &state,
                        enum uart_rx_parity parity)
```

Set the parity of the UART RX server.

The change of configuration takes place immediately.

```
void uart_rx_set_stop_bits(chanend c,
                           uart_rx_client_state &state,
                           unsigned stop_bits)
```

Set number of stop bits used by the UART RX server.

The change of configuration takes place immediately.

```
void uart_rx_set_bits_per_byte(chanend c,
                               uart_rx_client_state &state,
                               unsigned bits)
```

Set number of bits per byte used by the UART RX server.

The change of configuration takes place immediately.

2.4 Example Applications

2.4.1 GPIO Slice Examples

This uart is used in the `GPIO Com Port Demo` demo application to transfer commands from a host computer via uart to light the LEDs and read the adc on the Slicekit GPIO Slice Card. Refer to this item in xTIMEcomposer for more details on running this example application.

2.4.2 Basic Loopback Example

These modules can also be evaluated wired back to back using app_uart_back2back. To prepare the Slicekit core board to run this app connect wires between the 0.1" testpoints on the Triangle slot (or solder suitable headers on as shown in the picture below), such that ports 1A and 1E are connected. The headers corresponding to these ports are marked D0 and D12 respectively.

The app can then be built and run.

It will send the full set of characters from the UART TX, receive them on UART RX and then print them out in batches of 10 characters.

3 Simple UART

IN THIS CHAPTER

The intention of this module is to implement a high speed uart, at the expense of logical cores and 1-bit ports. Other modules provide lower-speed uarts that are logical core-efficient, or that may use fewer 1-bit ports. This module will support a 10 Mbaud rate with 100 MIPS logical cores, and correspondingly less with lower MIPS per logical core.

3.1 Resource Requirements

Resource	Usage
Code Memory	2110
Ports	2 x 1b
ChannelEnds	2
Logical Cores	2

3.2 Evaluation Platforms

3.2.1 Recommended Hardware

This module may be evaluated using the Slicekit Modular Development Platform, available from digikey. Required board SKUs are:

- XP-SKC-L2 (Slicekit L2 Core Board) plus XA-SK-XTAG2 (Slicekit XTAG adaptor) plus XTAG2 (debug adaptor), OR
- XK-SK-L2-ST (Slicekit Starter Kit, containing all of the above).

3.2.2 Demonstration Application

Example usage of this module can be found within the XSoftIP suite as follows:

- Package: sc_uart
- Application: app_uart_fast

3.3 API and Programming Guide

3.3.1 API

`void uart_rx_fast(in port p, streaming chanend c, int clocks)`

This function implements a fast UART.

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.

This function has the following parameters:

`p` input port, 1 bit port on which data comes in

`c` output streaming channel - read bytes of this channel (or words if you want to read 4 bytes at a time)

`clocks` number of clock ticks between bits. This number depends on the clock that you have attached to port p; assuming it is the standard 100 Mhz reference clock then clocks should be at least 10.

`void uart_tx_fast(out port p, streaming chanend c, int clocks)`

This function implements a fast UART.

It needs an unbuffered 1-bit port, a streaming channel end, and a number of port-clocks to wait between bits. It waits for a token on the streaming channel, and then sends a start bit, 8 bits, and a 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.

This function has the following parameters:

`p` output port, 1 bit port on which data should be transmitted

`c` input streaming channel - output bytes of this channel (or words if you want to output 4 bytes at a time)

`clocks` number of clock ticks between bits. This number depends on the clock that you have attached to port p; assuming it is the standard 100 Mhz reference clock then clocks should be at least 10.

3.3.2 Example Usage

Declare ports (and clock blocks if you do not want to run the ports of the reference clock):

```
clock refClk = XS1_CLKBLK_REF;

in port rx = XS1_PORT_1A;
out port tx = XS1_PORT_1B;
```

A function that produces data (just bytes 0..255 in this example)

```
void produce(streaming chanend c) {
    for(int i = 0; i < 256; i++) {
        c <: (unsigned char) i;
    }
}
```

A function that consumes data (and in this example throws it away)

```
void consume(streaming chanend c) {
    unsigned char buf[256];
    unsigned char foo;
    for(int i = 0; i < 256; i++) {
      c :> buf[i];
    }
    for(int i = 0; i < 256; i++) {
      printhexln(buf[i]);

    }
}
```

A main par that starts the logical cores:

```
int main(void) {
    streaming chan c, d;
    configure_out_port_no_ready(tx, refClk, 1);
    configure_in_port_no_ready(rx, refClk);
    clearbuf(rx);
    par {
        produce(d);
        consume(c);
        uart_tx_fast(tx, d, 10);
        uart_rx_fast(rx, c, 10);
    }
}
```

3.4 Example Application

These modules can be evaluated wired back to back using app_uart_fast. To prepare the Slicekit core board to run this app connect wires between the 0.1" testpoints on the Triangle slot (or solder suitable headers on as shown in the picture below), such that ports 1A and 1E are connected. The headers corresponding to these ports are marked D0 and D12 respectively.

The app can then be built and run.