

Future Technology Multicore

Hardware flow controlled UART Implementation for TC297B

Integration Guide  
(Revision A)

**Author(s):**  
Huy Dinh

**Release Date:**  
 06.07.2018

**ABSTRACT**

This document introduces the integration of a UART implementation with hardware flow control. Hardware flow control is used, if the computation of received data is slower than the datalink itself. The student car makes use of XBEE modules in transparent mode for communication from and to the car. The throughput of the modules is estimated to be about 3000 bytes/second, much less than the roughly expected 10000 bytes/second throughput of AURIX UART modules configured to run at 115200 Baud/s. This coupled with the fact that the XBEE module has a limited UART buffer, means that long transmissions can easily be congested without the AURIX being aware of it, resulting in a loss of data packets. Hardware flow introduces Request to Send (RTS) and Clear to Send (CTS) pins to signal a UART host (AURIX) when the buffer is ready for the next packet to be sent and received. The XBEE modules can use this to control the flow of UART bytes being sent by the AURIX until the buffer is clear.

LIST OF CONTENTS

[1 Principle of operation 1](#_Toc514184993)

[1.1 UART on the AURIX 1](#_Toc514184994)

[1.2 UART Hardware flow control on the AURIX 1](#_Toc514184995)

[1.3 UART Flow control with the XBEE modules 2](#_Toc514184996)

[2 Driver implementation 4](#_Toc514184997)

[2.1 UART Hardware Abstraction table 4](#_Toc514184998)

[2.2 UART configuration table 4](#_Toc514184999)

[2.3 Initializing UART channels 5](#_Toc514185000)

[2.4 Sending and receiving UART channels 5](#_Toc514185001)

[3 Driver API 6](#_Toc514185002)

[3.1 void UART\_init (); 6](#_Toc514185003)

[3.2 RC\_t UART\_ReadData (uartChannel\_t channel, uint8\_t\* data); 6](#_Toc514185004)

[3.3 RC\_t UART\_WriteData (uartChannel\_t channel, uint8\_t data); 6](#_Toc514185005)

[4 Integration into the overall system 7](#_Toc514185006)

[4.1 Integration and configuration of code files 7](#_Toc514185007)

[4.1.1 Files to be copied 7](#_Toc514185008)

[4.2 Usage 7](#_Toc514185009)

[4.2.1 Define the number of UART channels in use and their enumerators 7](#_Toc514185010)

[4.2.2 Define the entries inside the UART Hardware Abstraction Table 7](#_Toc514185011)

[4.2.3 Define the entries inside the UART Configuration Table 8](#_Toc514185012)

[4.2.4 Configure GPIO pins 9](#_Toc514185013)

[4.2.5 Interrupts (not here for now) 10](#_Toc514185014)

[4.2.6 Using the APIs 10](#_Toc514185015)

[5 Open issues 11](#_Toc514185016)

[6 Changelog 12](#_Toc514185017)

[7 List of abbreviations 13](#_Toc514185018)

[8 list of literature 14](#_Toc514185019)

# Principle of operation

## UART on the AURIX

Universal Asynchronous Receiver-Transmitter (UART) is an asynchronous communication protocol widely available across microcontrollers/microprocessors.

UART communication on the AURIX is facilitated by the Asynchronous/Synchronous Interface (ASCLIN) modules. These modules can be configured to run in various UART and SPI modes.

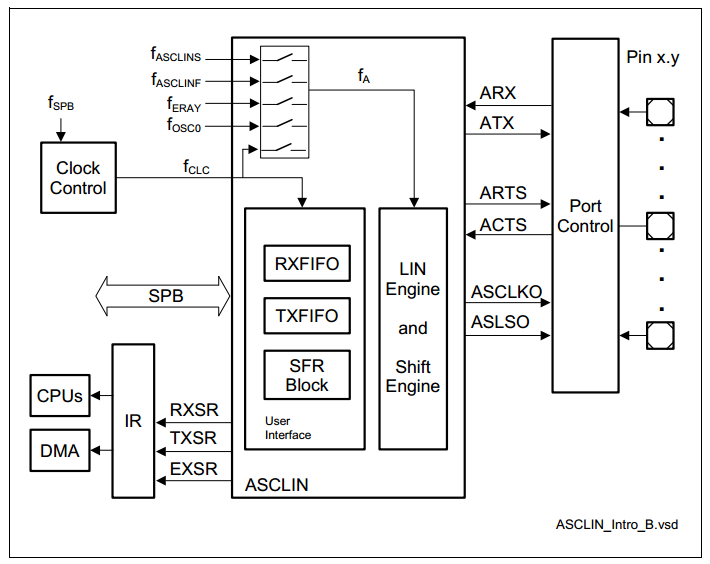


Figure 1 Block diagram of an AURIX ASCLIN module

## UART Hardware flow control on the AURIX

UART communication between two buffer-based can be augmented with a feature called flow control to ensure the synchronization of the two buffers, preventing data loss caused by overloaded buffers. Hardware flow control introduces two extra pins (RTS and CTS) in addition the original RX and TX pin.

In a generic setup, one client’s RTS will be connected to the other’s CTS pin and vice versa.

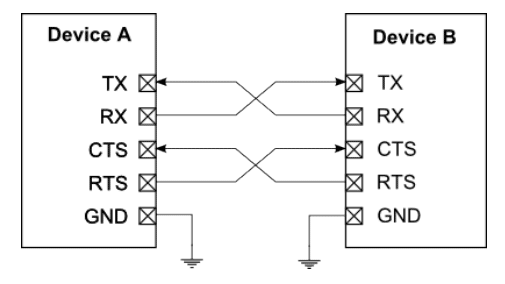


Figure Generic UART communication with hardware flow control

Hardware flow control is supported by the AURIX ASCLIN hardware used as a UART device.

The mechanism behind how the CTS and RTS pin work inside the module is visible in the architecture diagram below.

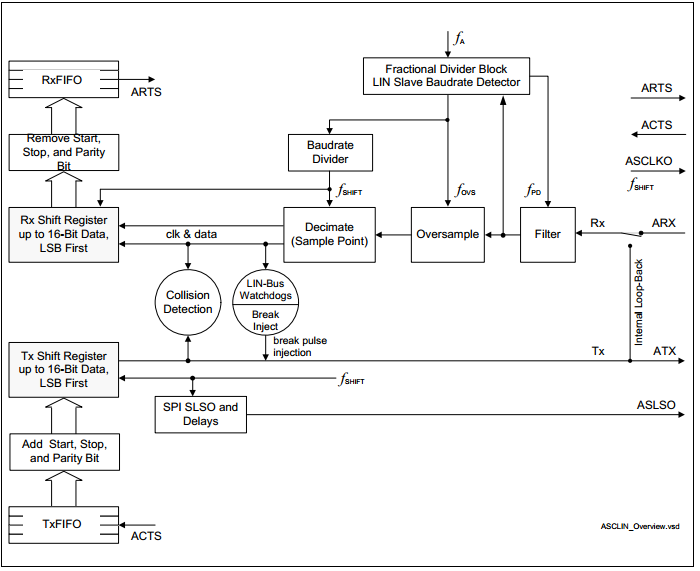


Figure 3 AURIX ASCLIN Architecture Overview

ARTS is an output pin connected to the Receipt (RX) FIFO buffer. ARTS will be asserted if the number of bytes inside the RX FIFO exceeds a certain size, and will be de-asserted if the fill level of the RX FIFO falls below another preset size. ARTS is not of interest in the scope of this implementation.

ACTS is an input pin connected to the Transmission (TX) FIFO buffer, and an inactive value of ACTS („Not Clear to Send“, logic level depending on the configuration) will prevent the data inside the FIFO from going into the Tx Shift Register to be shifted out through the ATX physical line.

## UART Flow control with the XBEE modules

The XBEE module flow control adds low-active CTS and RTS pins to its UART communication pins. Unlike the generic flow control setup, the CTS pin on the XBEE should be connected to the CTS pin on the host device (AURIX).

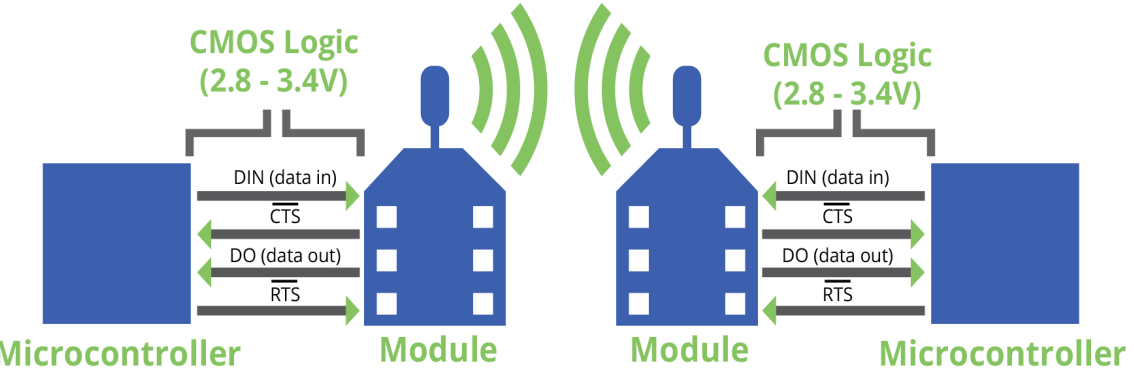


Figure 4 Flow controlled UART setup with the XBEE modules

If CTS is enabled on the XBEE side, when the DI buffer is 17 bytes away from being full, the XBEE module de-asserts CTS (sets it high) to signal to the host device to stop sending serial data. It reasserts CTS after the serial receive buffer has 34 bytes of space.

Therefore, by connecting the CTS pin on the XBEE to the ACTS pin on the AURIX with flow control enabled, we can stop the transmission when the XBEE UART buffer is full and resume when it is ready for reception again. This will prevent transmitted bytes from getting lost during transmission.

# Driver implementation

Since UART communication in the AURIX is facilitated by ASCLIN modules, which are also capable of SPI, the software implementation provides an abstraction layer to provide a “UART channel” concept. This is done through two configuration tables that can be modified.

The General-purpose Input/Output (GPIO) pins also need to be separately initialized with the right configurations in order to work with each ACLIN module. This is done by the GPIO module and is configurable in the GPIO Pin Initialization Table.

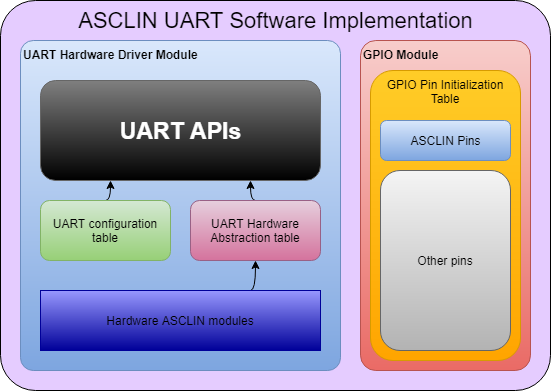


Figure 5 ASCLIN UART Software Implementation Overview

## UART Hardware Abstraction table

This table is an array of UART\_t named UART\_tbl and is used for primary configuration of a UART channel, including:

* Enabling/Disabling the channel.
* Selecting the ASCLIN module to handle the channel.
* Selecting the RX and CTS pin.
* Selecting the 3 Service Request Control (SRC) registers for interrupts: reception, transmission and error.

## UART configuration table

This table is an array of uartConfig\_t named UART\_Config and is used for UART-specific configurations including:

* Baud rate
* Data length
* Number of stop bits
* Parity type
* Enable/Disable CTS
* Whether CTS is Low active or High active
* Transmission FIFO Interrupt Level (The fill level of the TX FIFO that will trigger a TX interrupt).
* Reception FIFO Interrupt Level

## Initializing UART channels

When initializing the UART channels, the driver will go through all of the UART channel entries in the UART Hardware Abstraction table. If an entry is enabled, the ASCLIN module registered for that UART channel will be selected and initialized with respect to the configuration in the UART Configuration table. Interrupt hardware is also initialized.

## Adding flow control functionalities

Additional member variables were added to the UART configuration tables and the GPIO initialization table in order to allow for configuration of the CTS flow control pin.

* UART hardware abstraction: an input selection variable of type RxSelect\_t named CtsMux is added so that the user can select among the multiple CTS input pins allowed in an ASCLIN module.
* UART configuration table: ctsEnable (of enum type UART\_CtsEnable\_t) is added to enable/disable the use of CTS pin. ctsPolarity (of enum type UART\_CtsPolarity\_t) is added to select the polarity type of the CTS pin.
* GPIO initialization table: initialization of the CTS pin as a no-pull input needs to be added.

When called, UART\_init will go through both tables and configure the ASCLIN hardware according to the configurations, including these CTS configurations.

## Sending and receiving UART channels

The driver implementation makes use of the ASCLIN TX and RX FIFO buffers for transmission and reception of bytes respectively. This method allows for hardware buffering and flow control.

# Driver API

## void UART\_init ();

Initializes all UART (and their ASCLIN equivalent) modules with the configuration available in the UART configuration table if they are enabled.

## RC\_t UART\_ReadData (uartChannel\_t channel, uint8\_t\* data);

Read one byte from the RX FIFO of the selected UART channel.

Parameter channel: the UART channel whose RX FIFO the byte should be read from.

Parameter data: pointer to the memory position where the read data should be written to.

Return: RC\_SUCCESS if the reading operation succeeds, RC\_ERROR\_BUFFER\_EMPTY if the buffer is empty before the reading operation can take place, RC\_ERROR\_READ\_FAILS if the channel is disabled.

Note: UART\_init might need to be called earlier.

## RC\_t UART\_WriteData (uartChannel\_t channel, uint8\_t data);

Write one byte to the TX FIFO of the selected UART channel.

Parameter channel: the UART channel whose TX FIFO the byte should be written to.

Parameter data: the byte to be written.

Return: RC\_SUCCESS if the writing operation succeeds, RC\_ERROR\_BUFFER\_FULL if the TX buffer is full before the writing operation can take place, RC\_ERROR\_WRITE\_FAILS if the channel is disabled.

Note: UART\_init might need to be called earlier.

# Integration into the overall system

## Integration and configuration of code files

### Files to be copied

These files need to be copied to the project in these respective folders:

* UART.h to /src\_BSW/drv/UART
* UART.cpp to /src\_BSW/drv/UART
* UART\_config.h to /src\_BSW/drv\_config/UART
* UART\_config.cpp to /src\_BSW/drv\_config/UART

## Usage

### Define the number of UART channels in use and their enumerators

The user first needs to define the number of UART channels needed and the enumerators used as unique identification for each of these channel.

This can be done in **UART.h**. The number of UART channels can be defined by modifying value ò the macro UART\_CHANNELS\_NUM. Then the enumerators inside the enumeration uartChannel\_t needs to be filled up to the same number. These enumerators are used both as unique identifiers for the UART channels and the index inside the UART Hardware Abstraction Table and UART Configuration Table.

**#define** UART\_CHANNELS\_NUM 7

**typedef** **enum**

{

*uartusb* = 0,

*uart1*,

*uart2*,

*uart3*,

*uart4*,

*uart5*,

*uart6*

} uartChannel\_t;

Snippet 1 Defining the number of UART channels and their enumerators

In the example above, the system uses 7 UART channels, starting with **uartusb** (which has a value and index of 0) and ending with **uart6**.

### Define the entries inside the UART Hardware Abstraction Table

The UART Hardware Abstraction Table is an array of **UART\_t** named **UART\_tbl** declared and defined inside **UART.c**.

This array has a size of UART\_CHANNELS\_NUM and needs to be used to link ASCLIN modules and pin selection to the abstract UART channel. An example is shown below.

**const** UART\_t UART\_tbl[UART\_CHANNELS\_NUM] =

{

{ TRUE, &MODULE\_ASCLIN0, *RxSel\_a*, *RxSel\_a*, { &SRC\_ASCLIN0RX,

&SRC\_ASCLIN0TX, &SRC\_ASCLIN0ERR } },

{ FALSE, 0, (RxSelect\_t) 0, (RxSelect\_t) 0, { 0, 0, 0 } },

{ TRUE, &MODULE\_ASCLIN1, *RxSel\_g*, *RxSel\_a*, { &SRC\_ASCLIN1RX,

&SRC\_ASCLIN1TX, &SRC\_ASCLIN1ERR } },

{ FALSE, 0, (RxSelect\_t) 0, (RxSelect\_t) 0, { 0, 0, 0 } },

{ TRUE, &MODULE\_ASCLIN2, *RxSel\_b*, *RxSel\_a*, { &SRC\_ASCLIN2RX,

&SRC\_ASCLIN2TX, &SRC\_ASCLIN2ERR } },

{ FALSE, 0, (RxSelect\_t) 0, (RxSelect\_t) 0, { 0, 0, 0 } },

{ TRUE, &MODULE\_ASCLIN3, *RxSel\_e*, *RxSel\_a*, { &SRC\_ASCLIN3RX,

&SRC\_ASCLIN3TX, &SRC\_ASCLIN3ERR } },

};

Snippet 2 Example UART Hardware Abstraction Table

These configuration can be seen from the example above (in addition to the enumeration defined in the last chapter)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **UART channel** | **Enabled** | **ASCLIN Module** | **RX pin selection** | **CTS pin selection** | **Interrrupt register set** |
| uartusb | True | ASCLIN0 | A | A | ASCLIN0 |
| uart1 | False | N/A | N/A | N/A |  |
| uart2 | True | ASCLIN1 | G | A | ASCLIN1 |
| uart3 | False | N/A | N/A | N/A |  |
| uart4 | True | ASCLIN2 | B | A | ASCLIN2 |
| uart5 | False | N/A | N/A | N/A |  |
| uart6 | True | ASCLIN3 | E | A | ASCLIN3 |

Since the RX and CTS pins are input pins used in the ASCLIN module, the module can select which pins should be used for the respective operations. These selections range from RxSel\_a (0) to RxSel\_h (7) (defined in the enumeration **RxSelect\_t** in **PORT.h**). Please refer to the TC29X User’s Manual for more information about the hardware pins associated with these selections.

### Define the entries inside the UART Configuration Table

This table is an array of **uartConfig\_t** named **UART\_Config** defined in **UART\_Config.c** and has a size of **UART\_CHANNELS\_NUM.**

This table must be used to define UART-specific characteristics, following the format provided by **uartConfig\_t.** An example is found below.

uartConfig\_t **const** UART\_Config[UART\_CHANNELS\_NUM] =

{

{*uart\_115200* , *\_\_8*, *StopBit\_1*, *None*, *CtsDisable*, *LowActive*, *TxInt\_1* , *RxInt\_1*},

{*uart\_115200* , *\_\_8*, *StopBit\_1*, *None*, *CtsDisable*, *LowActive*, *TxInt\_1* , *RxInt\_1*},

{*uart\_115200* , *\_\_8*, *StopBit\_1*, *None*, *CtsDisable*, *LowActive*, *TxInt\_1* , *RxInt\_1*},

{*uart\_115200* , *\_\_8*, *StopBit\_1*, *None*, *CtsDisable*, *LowActive*, *TxInt\_1* , *RxInt\_1*},

{*uart\_115200* , *\_\_8*, *StopBit\_1*, *None*, *CtsEnable*, *LowActive*, *TxInt\_1* , *RxInt\_1*},

{*uart\_115200* , *\_\_8*, *StopBit\_1*, *None*, *CtsDisable*, *LowActive*, *TxInt\_1* , *RxInt\_1*},

{*uart\_115200* , *\_\_8*, *StopBit\_1*, *None*, *CtsDisable*, *LowActive*, *TxInt\_1* , *RxInt\_1*}

};

Snippet 3 Example UART Configuration Table

These configurations below can be seen from the UART configuration table above

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **UART channel** | **Baud rate** | **Byte length** | **Number of stop bits** | **Parity** | **CTS** | **CTS logic** | **TX Interrupt level** | **RX Interrupt level** |
| uartusb | 115200 | 8 bits | 1 | None | Disabled | Low Active | 1 | 1 |
| uart1 | 115200 | 8 bits | 1 | None | Disabled | Low Active | 1 | 1 |
| uart2 | 115200 | 8 bits | 1 | None | Disabled | Low Active | 1 | 1 |
| uart3 | 115200 | 8 bits | 1 | None | Disabled | Low Active | 1 | 1 |
| uart4 | 115200 | 8 bits | 1 | None | Enabled | Low Active | 1 | 1 |
| uart5 | 115200 | 8 bits | 1 | None | Disabled | Low Active | 1 | 1 |
| uart6 | 115200 | 8 bits | 1 | None | Disabled | Low Active | 1 | 1 |

For more information on the Interrupt levels, please refer to the TC29X User’s Manual.

### Configure GPIO pins

GPIO pins need to be correctly configured in order to work with the ASCLIN modules. This can be done by modifying the port configuration table inside **PORT\_cfg.c** (An array of **portPin\_cfg\_t** named **PORT\_config**). Up to 3 pins need to be configured: TX, RX and CTS. Please check the TC29X User’s Manual for the pins capable of these functions for each ASCLIN module.

{(Ifx\_P \*)&MODULE\_P02, 0,*\_outputPushPullAlt2*,*PES\_inactive*,*\_cleared*, *cmosAutomotiveSpeed1*},

Snippet 4 Example UART TX pin configuration

In the example above, the following configuration for Pin 0 Port 2 can be observed:

* Configured as a Push Pull Output pin, Alternative Function 2 (To be used as TX pin for the UART, check the TC29X User’s Manual for more information).
* Not connected to PES
* Pin is cleared after initialization
* CMOS Automotive Speed 1

{(Ifx\_P \*)&MODULE\_P02,1,*\_inputPullUp*, *PES\_inactive*,*\_cleared*, *cmosAutomotiveSpeed1*},

Snippet 5 Example UART RX pin configuration

In the example above, the following configuration for Pin 1 Port 2 can be observed:

* Configured as a pulled up input
* Not connected to PES
* Pin is cleared after initialization
* CMOS Automotive Speed 1

Since the ASCLIN module will select this pin as RX during UART initialization, there is no need to specify the alternative function for this pin in the GPIO initialization.

{(Ifx\_P \*)&MODULE\_P10,7,*\_inputNoPullDevice*, *PES\_inactive*,*\_cleared*, *cmosAutomotiveSpeed1*},

Snippet 6 Example UART CTS pin configuration

In the example above, the following configuration for Pin 7 Port 10 can be observed:

* Configured as an input with no pull registers
* Not connected to PES
* Pin is cleared after initialization
* CMOS Automotive Speed 1

### Interrupts

Each ASCLIN module has 3 interrupt sources, represented by 3 Service Request Control (SRC) registers (x being the number of the ASCLIN module):

* SRC\_ASCLINxRX: reception interrupt
* SRC\_ASCLINxTX: transmission interrupt
* SRC\_ASCLINxRX: error interrupt

By default, the UART\_init function will prepare the ASCLIN hardware for interrupts. However, these interrupts will not be usable without being correctly configured by the Interrupt module. Please refer to the Interrupt integration guide for more information on how to configure ASLCIN interrupts.

### Using the APIs

After all of the configurations above are done properly, the APIs are ready to be used for UART communication.

Before any other API can be called, UART\_Init() needs to be called. After that, UART\_WriteData and UART\_ReadData can be called to write and read to UART buffer respectively.

uint8\_t readSample;

UART\_init();

// Write 0xAA to the TX buffer of uart4

UART\_WriteData(*uart4*, 0xAA);

// Read a byte from the RX buffer of uart6 and save it onto readSample

UART\_ReadData(*uart6*, &readSample);

Snippet 7 Example uses of the API

# Open issues

# Changelog

# List of abbreviations

|  |  |  |
| --- | --- | --- |
| API | - | Application Programming Interface |
|  |  |  |
|  |  |  |

# list of literature

|  |  |
| --- | --- |
| [1] | Infineon, *TC29x B-Step User’s Manual v1.3,* 2014. |

**APPENDIX**

<Anything which is relevant but too long to be put directly into the document. You can use this area to add code.>