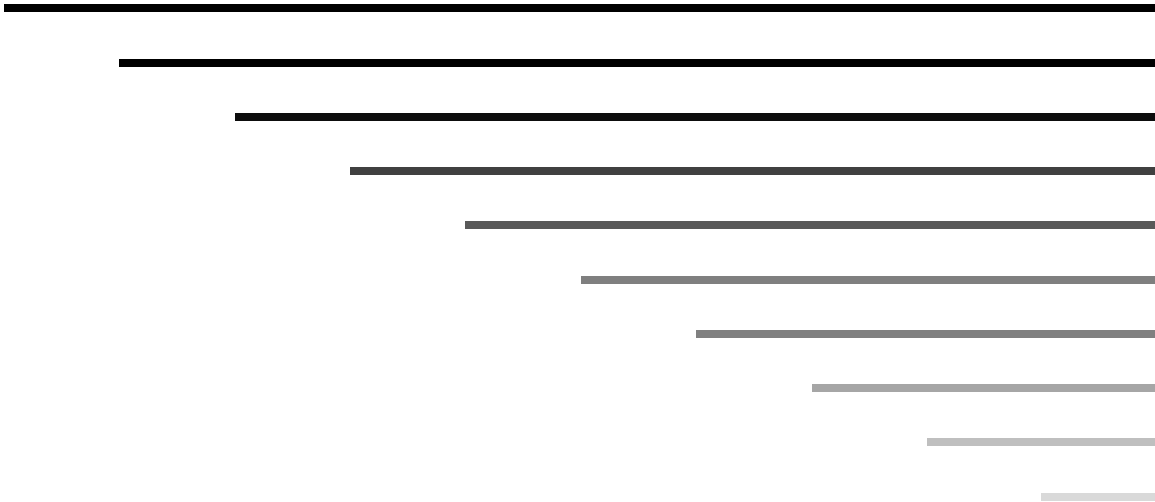


*Embedding it better...*



μTasker Document

**μTasker – UART User's Guide**



## Table of Contents

1. Introduction.....	3
2. UART Characteristics.....	3
2.1. Additional Control Lines .....	5
3. Initialisation .....	6
3.1. Notes about UART use in the μTasker demo project .....	8
4. Reception .....	8
5. Transmission .....	9
6. Transmitter Blocking.....	10
7. Flow Control .....	11
8. RS485 Operation .....	14
9. UART as Debug Output .....	18
10. DMA Operation.....	19
11. Break Condition.....	20
12. Message Mode.....	20
13. Echo Mode .....	20
14. Escape Operation .....	21
15. Inter-Space Timing .....	21
16. Multi-Drop, 9-Bit Mode.....	22
17. Changing Operation Modes and Flushing Queues.....	24
18. UARTs in the μTasker Simulator .....	24
19. External UARTs .....	24
20. Conclusion.....	25
Appendix A – Hardware Dependencies .....	26
a) UART Peripheral Pins .....	26
b) RS485 mode .....	26

## 1. Introduction

Most processors include one or more UARTs (Universal Asynchronous Receiver/Transmitter) as standard internal peripherals. The interfaces are extremely popular for various communication protocols for inter-chip or inter-board communication and also for interaction with users via a terminal emulator program.

The UART interface, also known as serial interface, in the µTasker project allows all UARTs that a processor owns to be simply configured and utilised via a standardised but versatile programming interface. This interface, along with various operating modes such as DMA, RS485 and flow control, is described in this document together with practical code examples.

## 2. UART Characteristics

UARTs are asynchronous, which means that the data is not synchronised by using a clock signal. This requires that the receiver and transmitter in a point-to-point connection, or on a bus, are configured with the correct speed and other parameters match. The speed should be accurate enough to ensure that the receiver, when synchronised to an individual character, doesn't drift so much that it can result in errors recognising the final bits in the character.

Exactly what speed (Baud Rate) tolerance can be accepted depends on the absolute accuracy of transmitter and receiver; for example if both are inaccurate by +x% the connection will still operate correctly since the relative accuracy is still high. Generally a good rule for the required accuracy of transmitters and receivers that are not related in any way is maximum 1% tolerance for the transmitter and a receiver than can accept up to a maximum of 2% tolerance.

When measured at the processor output pin, an idle line is represented by a logic '1' level. Often an RS232 driver is used to drive the external line, which converts a logic '1' to a negative signal (-3V..-15V, but typically around -9V as delivered by common driver chips supplied by logic voltage power supplies of 3V3 or 5V). The following discussion will always reference the signals at the processor pins, which is thus inverted to the signal polarity when measured on the RS232 line.

A UART character starts always with a single start bit. The transmitter sends a low level signal for a single bit period. This period depends on the speed of the UART (the Baud Rate) and the bit period is given by  $(1/\text{Baud Rate})\text{s}$ . For example, if the Baud Rate is set to 19'200 a single bit period (and thus the start bit timing) will be 5120.8333us +/- 1%.

The data follows the start bit and is sent always LSB (Least Significant Bit) first. For example, the data value 0x01 will be sent as '1', '0', '0', '0', '0', '0', '0', '0', assuming that the data is 8 bits in length. Each data bit is one bit period in length.

Following the data content there is an optional parity bit. If present, this can be even parity, odd parity, '1' parity or '0' parity and is also present for one bit period.

Each character is completed by a number of stop bits. Stop bits have the level '1' and are typically, 1, 1.5 or 2 bit period in length.

In the case of a setting with 8 bit characters, one parity bit and 2 stop bits the maximum data bit throughput is the  $(\text{Baud Rate} * 8) / 12$ ; 19'200Baud has this a maximum data bit rate of 12'800 bits or 1'600 bytes per second.

A following character is recognised by the next start bit detected on the line. The spacing from the end of the last stop bit to the start of the next start bit is not defined at the physical layer and is also not important for the operation of the receiver since, being asynchronous, it resynchronises to the following character without any relationship to what was previously received on the line.

The receiver typically operates by sampling the state of the Rx signal on the line at a faster rate than the Baud Rate clock. It is usual to use a 16x oversampling clock so that the falling edge of start bit can be identified with an accuracy of 1/16 the bit period. Once the start bit has been detected the clock is synchronised to the new character and each bit of the character can be sampled 8 oversampling clock periods later. As long as the sampling clock (the receive Baud Rate setting) is adequately accurate compared to the absolute Baud Rate of the received signal it will remain accurate over the duration of a single character. On detection of the start bit of the following character the process can begin again ensuring that there is no relationship to the previous received character (which could arrive for a different source on a bus system) and no inaccuracies will accumulate. Many chips also perform multiple sampling of each bit and in order to improve performance in noisy environments; typically this is achieved by sampling not just once at the 8<sup>th</sup> oversampling clock point but three times – at 7, 8 and 9 clock points. The logic level detected will be the one that is sampled at least twice by the three sample points and a single incorrect state due to impulse noise during the sample phase will be ignored.

Figure 1 illustrates a typical character as discussed in this section, where 1.5 stop bits and even parity are shown as reference. The data being transmitted is the 8 bit value 0xa1 and the Baud Rate is 19'200.

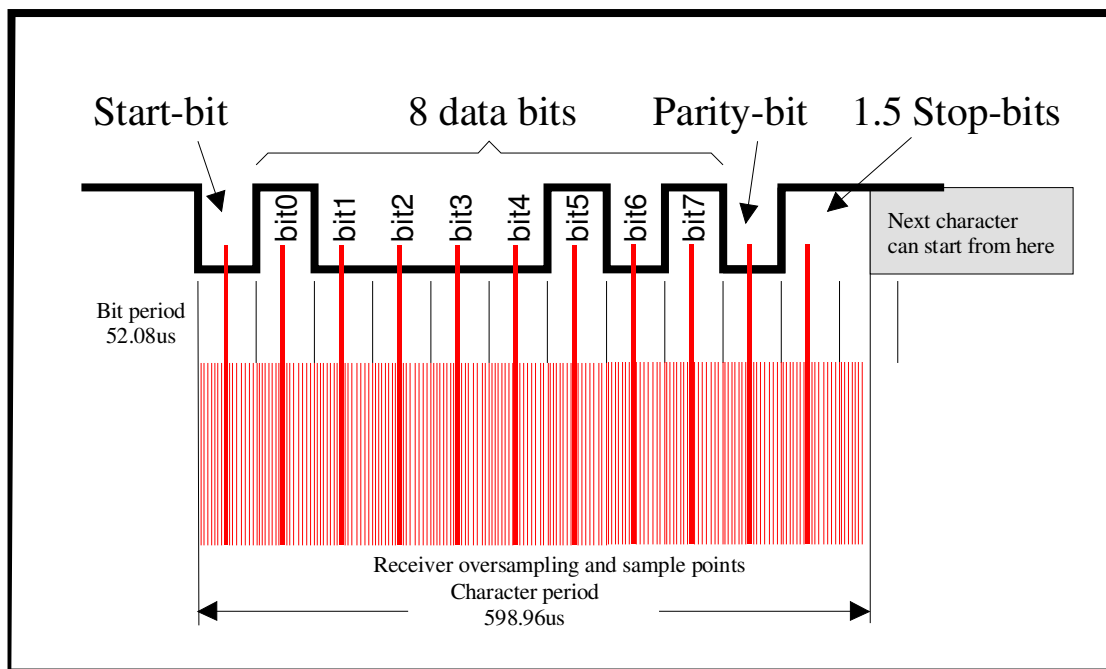


Figure 1 – Example of an asynchronous character

## 2.1. Additional Control Lines

The simplest bi-directional UART connection consists of three signals.

- TXD – the transmitted data
- RXD – the received data
- Ground – the reference ground

The TXD at one end of the connection is connected to the RXD at the other end.

In some applications additional control lines are required, which are also specified in the EIA RS-232-C standard. Furthermore additional protective grounds may be found in connection cables too. When discussing these it is often useful to reference the two ends of the connection as the Data Terminal Equipment (DTE) and the Data Circuit-terminating Equipment (DCE). The easiest method of imaging this is to image a PC connected to an analogue modem; the PC is the DTE and the modem is the DCE where data transmission originates from the PC (it drives the TXD line) and reception data arrives at the PC (it receives via RXD). The commonly used control signals and their directions are thus as follows, whereby often not all are used and their pin position on a connector varies between connector types!

- RTS – Request To Send (DTE → DCE)
- CTS – Clear To Send (DTE ← DCE)
- DTR – Data Terminal ready (DTE → DCE)
- DSR – Data Set Ready (DTE ← DCE)
- DCD – Carrier Detect (DTE ← DCE)
- RI – Ring Indicator (DTE ← DCE)

The most commonly used line are RTS and CTS in conjunction with hardware based flow control, where two devices are connected via RXD ↔ TXD and RTS ↔ CTS. The receiver uses its CTS line to signal that its receiver can accept data and can stop the flow of data by negating this signal. The other transmitter immediately stops sending further data until the CTS is asserted again.

This method of flow control and also a character bases one called XON/XOFF are described later in this document.

### 3. Initialisation

To activate support for the serial interface in the µTasker project ensure that the define `SERIAL_INTERFACE` is enabled in `config.h`.

Before a UART can be used for transmission and/or reception in the µTasker project it needs to be initialised. This initialisation, or configuration, takes place when the serial interface is opened. Each UART which the processor supports can be opened individually so that each UART interface can be used independently.

Code 1 show UART1 being opened, which assumes that the processor has at least 2 UARTS (the UARTs are always counted 0, 1, 2 etc.).

```

QUEUE_HANDLE fnOpenUART(void)
{
    TTYTABLE tInterfaceParameters;           // table for passing information to driver
    QUEUE_HANDLE SerialPortID;               // UART handle to be obtained during open
    tInterfaceParameters.Channel = 1;        // set UART channel for serial use
    tInterfaceParameters.ucSpeed = SERIAL_BAUD_19200; // baud rate 19'200
    tInterfaceParameters.Rx_tx_sizes.RxQueueSize = 256; // input buffer size
    tInterfaceParameters.Rx_tx_sizes.TxQueueSize = 512; // output buffer size
    tInterfaceParameters.Task_to_wake = TASK_APPLICATION; // wake task on rx

#ifdef SUPPORT_FLOW_HIGH_LOW
    tInterfaceParameters.ucFlowHighWater = 80; // set the flow control high in %
    tInterfaceParameters.ucFlowLowWater = 20; // set the flow control low in %
#endif

    tInterfaceParameters.usConfig =
        (CHAR_8 + NO_PARITY + ONE_STOP + USE_XON_OFF + CHAR_MODE);

#ifdef SERIAL_SUPPORT_DMA
    tInterfaceParameters.ucDMAConfig = UART_TX_DMA; // activate DMA on transmission
#endif
    if ((SerialPortID = fnOpen( TYPE_TTY, FOR_I_O, &tInterfaceParameters )) != 0) {
        // open the channel with defined configurations (initially inactive)
        fnDriver(SerialPortID, ( TX_ON | RX_ON ), 0 ); // enable rx and tx
    }
    return SerialPortID; // return the serial port handle for this UART
}

```

Code 1 - example of UART initialization

The UART characteristics, such as Baud Rate, Stop bits, Parity, etc. are passed to the open routine as a `TTYTABLE` structure. The UART as specified by the Channel parameter will be initialised with the defined configuration.

The UART interface is buffered and the size of the transmitter and receiver buffers can be defined individually as show in the example. In this case the receiver will have a buffer size of 256 bytes and the transmitter a buffer size of 512 bytes. This buffer space is created on heap memory during the open sequence. The receive buffer can hold received data until read by application software and application software can pass data to be transmitted up to the transmit buffer size, after which the UART driver will ensure that it is completely transmitted over the UART interface.

The optional settings for flow control and DMA are discussed later in the document; they are shown here only for completeness. *If options are enabled but the capabilities not used their values should be set to 0 and not left as undefined values otherwise they could lead to unpredictable behaviour.*

The generic call to `fnOpen()` initialises the UART and a serial port handle is returned. This reference of type `QUEUE_HANDLE` is used later for all further interaction with this UART.

The open call is show again below with more details concerning its parameters:

```
SerialPortID = fnOpen(
    TYPE_TTY,           // serial interface driver type (UART)
    FOR_I_O,           // for use as input/output device (Rx/Tx)
    &tInterfaceParameters); // configuration details
```

Code 2 - the `fnOpen()` function used to open a serial interface

Note that `TYPE_TTY` must be used for a UART.

The example configures for both reception and transmission (`FOR_I_O`). If a UART is required one to operate as a receiver or a transmitter the following parameters can be used instead, in which case no unnecessary buffer will be defined for the non-used direction.

- `FOR_READ`
- `FOR_WRITE`

The receiver and transmitter have been configured by the `fnOpen()` call, as long as it doesn't return a zero handle (this usually means that there is not adequate heap to define the requested buffer sizes) but they haven't been activated. To change the state of the receiver and/or transmitter the `fnDriver()` function is used. This function uses the serial interface handle to specify which interface is to be acted on.

```
fnDriver(
    SerialPortID,           // the serial interface handle
    ( TX_ON | RX_ON ),     // enable rx and tx
    0 );                   // unused parameter in this case
```

Code 3 - the `fnDriver()` function used to enable receiver and transmitter

After enabling the transmitter and receiver the UART is ready for use. In this example the UART has been opened in character mode (see parameter `CHAR_MODE` in `tInterfaceParameters.usConfig`). This is the most common mode of operation which means that each received character is considered to be a complete message. Each time a character is received at the UART the UART's owner task, as specified by `tInterfaceParameters.Task_to_wake = OWN_TASK_APPLICATION;` is woken. This task can then read the UART input to treat the received character(s).

The following section shows how the read process operates. It also shows the case when operating in character mode. Further modes of operation as discussed in later sections of the document.

### 3.1. Notes about UART use in the µTasker demo project

If the serial interface is enabled in the µTasker demo project the UART, as specified by `DEMO_UART` in `app_hw_XXXX.h` (where `XXXX` is the target processor for the project), is used as a command line interface. Its parameters are determined by the default setting in `cParameters[]` in `application.c`.

The UART used as demonstration interface can thus be adapted by changing the UART number, for example:

```
#define DEMO_UART      2                // use UART 2 for demo interface
```

To disable this UART for use for alternative purposes comment out `DEMO_UART`. This will remove all code working with the serial interface but keep general UART driver support active.

Some processors allow multiplexing UARTs to various processor pins. See Appendix A for details of processor specific configuration of the UART peripheral pins .

## 4. Reception

This section shows how received data is read from the UART. It assumes that the UART is configured to operate in character mode. Further operating modes are discussed in later sections of this document. See section 3 for an example of initialising the UART interface to character mode.

```
extern void fnApplication(TTASKTABLE *ptrTaskTable)
{
    unsigned char ucInputByte;
    while (fnRead( SerialPortID, &ucInputByte, 1) != 0) {
        while (ucInputByte-->0) {
            toggle_output();
        }
    }
}
```

Code 4 - a task receiving characters from a UART with handle `SerialPortID`

Code 4 shows a simple task which has been configured (using `fnOpen()`) to be the owner task for the serial interface with handle `SerialPortID`. The task is woken on every character reception from the UART and is shown reading the input queue one byte at a time until there are no more input characters remaining in the queue. Depending on the value of each byte received it then toggles and output this number of times (*simple example*).

All reads are performed using the generic `fnRead()` function where, following the handle to the interface, a buffer is specified to read the specified amount of input data to. The function returns the number of characters copied to the buffer and 0 if there were no more waiting.

Code 5 shows a further variation which first uses `fnMsgs()` to check how much data is waiting in the input buffer before reading this exact amount into a buffer.

```
extern void fnApplication(TTASKTABLE *ptrTaskTable)
{
    QUEUE_TRANSFER length;
    unsigned char ucInputBuffer[100];
    while ((length = fnMsgs(SerialPortID)) != 0) { // get number of waiting bytes
        if (length > sizeof(ucInputBuffer)) { // ensure buffer can't be overrun
            length = sizeof(ucInputBuffer);
        }
        fnRead(SerialPortID, ucInputBuffer, length); // read to a local buffer
    }
}
```

Code 5 - a task filling a buffer from a UART with handle SerialPortID

`fnMsgs()` return the amount of waiting messages. In character mode each character is counted as a single message and so it returns the number of waiting characters.

## 5. Transmission

This section shows how data is set to the UART interface. See section 3 for an example of initialising the UART.

Transmission is controlled using the generic `fnWrite()` function as shown in code 6.

```
fnWrite(SerialPortID, (unsigned char *)"1234567890", 10);
```

Code 6 - transmission of a string to a UART with handle SerialPortID

A write causes the data to be first copied to the output buffer of the serial interface. The size of this buffer is defined when the interface is opened. The routine returns the amount of data copied to the output buffer. If more data is passed than the buffer can accept, the additional data will be ignored.

The first write will cause buffer transmission to commence and each character will be transmitted by interrupt or by DMA (if the processor supports DMA and the DMA transmission mode has been configured).

Further writes during active transmission will be queued in the output buffer for transmission as soon as previous data bytes have been sent.

In some applications it is critical to be able to ensure that no transmit data is lost due to lack of output buffer space. The application can check the available buffer space before attempting a write by calling the `fnWrite()` function with a zero data pointer. The return value is the amount of space left in the output buffer if the given amount of data were to be copied and will only be 0 if there is not adequate space for the complete amount of data to be accepted (*1 will be returned if the available space is the same as the length requested*).

If there is not adequate space to perform a write of the complete data the application should wait until the output buffer is emptied using the `TX_FREE` event as detailed in the following section.

## 6. Transmitter Blocking

Since the UART's output buffer may often be filled by software faster than data can be transmitted the application may need to stop sending additional data when the output buffer becomes critically full. Rather than polling the remaining space in the output buffer the `TX_FREE` event is foreseen as a mechanism to inform the application when the buffer content has reduced to a low level; *the value used is the same as the low-water level of the UART's receiver*. The event is activated by entering the task to be woken by this transmitter event, which is a single-shot event, meaning that it needs to be re-activated each time that it should be reported.

```

if (fnWrite(SerialPortID, 0, 10) == 0) {
    fnDriver(SerialPortID, MODIFY_WAKEUP, (MODIFY_TX | OWN_TASK));
                                                // enable TX_FREE event
}
else {
    fnWrite(SerialPortID, (unsigned char *)"1234567890", 10);
}

```

Code 7 – check of output buffer space before transmission

Code 7 shows the remaining output buffer space first being checked before sending data. If there is adequate space, the data is immediately written. If the output buffer doesn't have adequate space to accept the complete data, the transmitter event if enabled by entering the task name.

When the amount of data in the output buffer reduces to, or below, the low-water value the `TX_FREE` event will be sent to wake up the task. In many cases the event doesn't have to be handled specifically, although it should be read, since a repeat of the code in example 7 will then result in the data being copied. Code 8 shows an excerpt from the command line menu interface in the µTasker demo project (from `debug.c`) which uses the `TX_FREE` event to continue transmission of a menu listing. In this case it reacts specifically to the `TX_FREE` event to coordinate the process. Using this method it is possible to transmit a large amount of menu items using only a small output buffer size since `TX_FREE` is used as an event to add further data to the buffer until the transmission has completed.

```

extern void fnDebug(TTASKTABLE *ptrTaskTable)
{
    QUEUE_HANDLE PortIDInternal = ptrTaskTable->TaskID; // queue ID for task input
    unsigned char ucInputMessage[SMALL_MESSAGE]; // space for receiving messages
    while ( fnRead( PortIDInternal, ucInputMessage, HEADER_LENGTH)) { // read input
        switch ( ucInputMessage[ MSG_SOURCE_TASK ] ) { // switch on source
            case INTERRUPT_EVENT:
                if (TX_FREE == ucInputMessage[MSG_INTERRUPT_EVENT]) {
                    fnDisplayHelp(iMenuLocation); // continue sending further menu items
                }
                break;
        }
    }
}

```

Code 8 – Example of specific handling the `TX_FREE` event

## 7. Flow Control

Flow control is important when a serial receiver cannot always keep up with the data being sent by a transmitter. This is not necessarily due to the fact that the receiver may be slower at handling the received data but may also be due to the fact the received data cannot be processed as fast as it is received. For example, if data is being received at a rate of 19'200Baud on UART 0 and is being sent to UART 1 with a rate of 9'600 Baud it will not be able to indefinitely receive and store the data arriving at UART0. The point at which the receiver must request the remote transmitter to pause depends on the size of its input buffer. Once the input buffer gets critically full then it is time to do this. Once the input buffer has adequate space again the data transfer can be started again.

Two types of flow control are used:

- **In-channel flow control** – XON/XOFF is typically used on ASCII encoded connections. This doesn't require additional control lines but needs to reserve special characters for the purpose (XOFF = 0x13 – this can be sent from a terminal with CTRL + S / XON = 0x11 – this can be sent from a terminal with CTRL + Q).
- **Hardware Flow control** – this uses typically RTS and CTS control lines.

Figure 2 shows a configuration which allows simple testing of flow control operation. It works with either XON/XOFF or RTS/CTS control as illustrated by the timing diagram in figure 3. Furthermore the complete flow can be paused by sending an XOFF from the terminal monitoring the third UART's output (CTRL + S). The operation can then be restarted by removing the stop condition with XON (CTRL + Q). UARTs 1 and 2 need to be configured to both use the same serial settings, including the same flow control method.

Practical tests of this configuration are simple on processors with three UARTs. UART2 can also be on a second board when testing processors with just 2 UARTs.

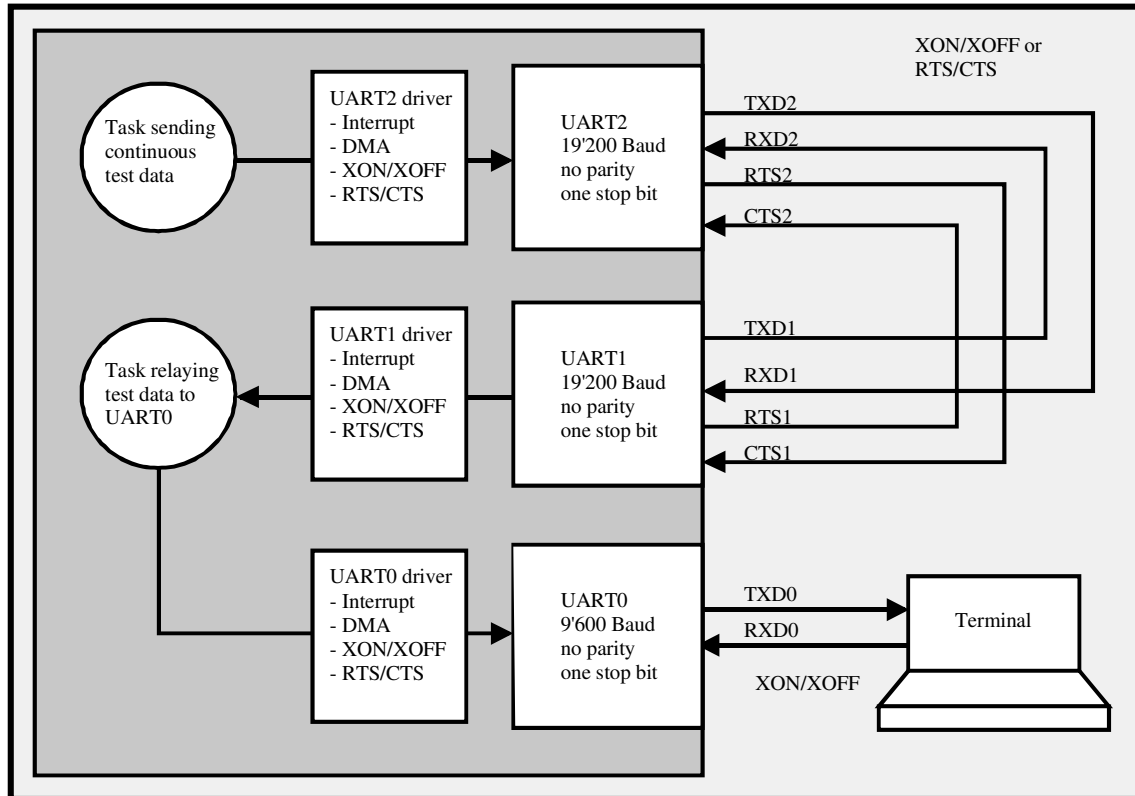


Figure 2 – Flow Control test configuration (to test XON/XOFF and RTS/CTS control in interrupt or DMA driven modes)

Since the speed of UART0 is only half that of UARTs 1 and 2, it represents a bottle-neck in the system. It is assumed that UART0 transmitter has only a small buffer and so the data transmitted from UART2 has to be buffered at the receiver of UART1. UART1 input buffer fills faster than it can be emptied and at some point (depending on the size of the buffer and the level of its “high-water” mark) it will become critically full. To avoid data being received that cannot be put into the buffer (overrun), the flow control kicks in; in XON/XOFF mode the driver of UART1 will send an XOFF; in RTS/CTS mode the driver of UART1 will negate its RTS line. The flow control action will stop the transmitter from sending any more data. Note that it may take a short time for the transmitter to actually stop sending its data and so the high-water level is set at a level where the receiver can still accept enough data to avoid an overrun at its input before the transmitter finally stops. In systems where it is known that the reaction is fast, the high water mark can be set close to the maximum buffer level (say 95% and higher).

Flow control has high priority due to its function. When operating in XON/XOFF mode, if the UART1's transmitter is busy sending a message from its output buffer when the flow control needs to be stopped, the XOFF character will immediately be sent out on to the TXD1 line without first waiting for the transmit buffer to be emptied. The flow control characters can thus be sent in the body of transmitted messages.

When the data flow from TXD2 to RXD1 has been stopped the TXD2 line becomes idle. In the test configuration this gives the relay task the opportunity to continue sending data to the UART0 transmitter, thus reducing the data content at the UART1 input buffer. Once the buffer content level reduces to the “low-water” mark the flow will again be enabled; in XON/XOFF mode the driver of UART1 will send the XON character; in RTS/CTS mode the UART1 driver will activate its RTS output. UART2 transmitter is thus allowed to send again.

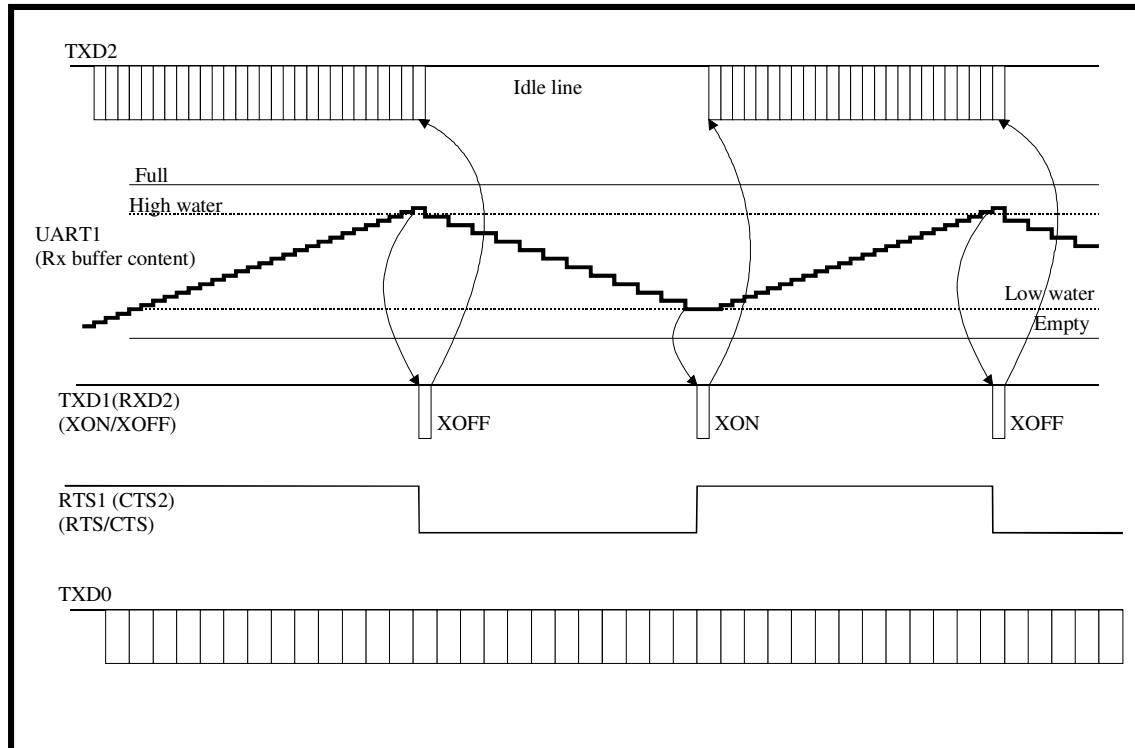


Figure 3 – Flow control operation in the test configuration, controlled by the high and low water marks in the receive buffer of UART1

The flow control process repeats, following the pattern illustrated in figure 3, resulting in a continuous and uninterrupted flow of data from the transmitter of UART0. The throughput is regulated by the input buffer of UART1 in a manner that its input buffer is neither in risk of overflowing (safety margin defined by the “high-water” mark) nor in risk not having data to supply the UART0 output (safety margin defined by the “low-water” mark).

If the flow control at the output of UART0 is paused, by the terminal sending an XOFF character, the transmission from UART0 is stopped immediately and the data flow from UART2 to UART1 will be stopped as soon as the input of UART1 becomes critically full. In this case all transmitters will remain in the idle state until the terminal sends an XON character, after which the transmission to the UART0 output will immediately continue and the cyclic flow control as depicted in figure 3 recommence.

## 8. RS485 Operation

The RS485 bus can consist of a semi-duplex two wire electrical connection or a four wire full duplex connection. There can also be multiple devices sharing the bus.

When a device transmits on the RS485 bus it may need to drive its output transceiver. For this reason in the RS485 mode the RTS line is activated when transmitting and deactivated once the transmission of a single message frame has completed.

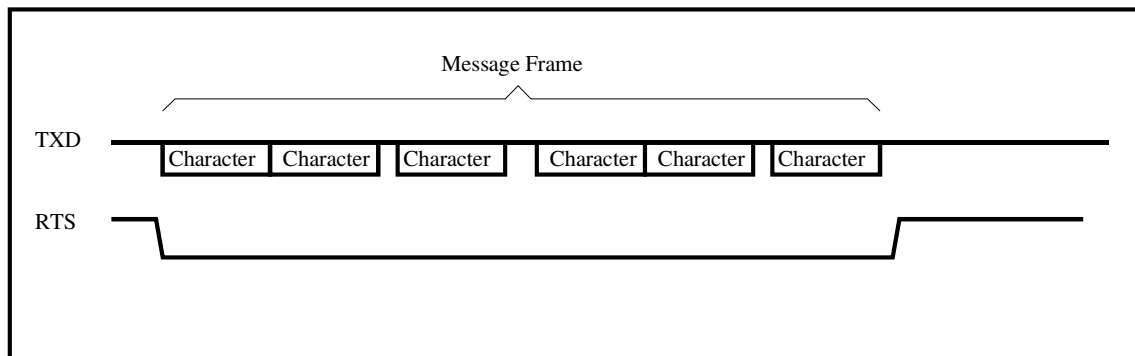


Figure 4 RTS control during a single frame

As illustrated in figure 4 the RTS signal is asserted at the start of the first character of a message frame. The particular frame consists of a number of characters which are each transmitted asynchronously, meaning that there can theoretically also be a short gap between each character. The RTS remains active until the end of the final stop bit of the last character in the frame, after which it is de-asserted as quickly as possible to avoid the bus being unnecessarily driven after the message transmission has terminated. (A de-assertion of the RTS during the transmission of the final stop bit is generally acceptable since the idle state of the bus corresponds to the same state.)

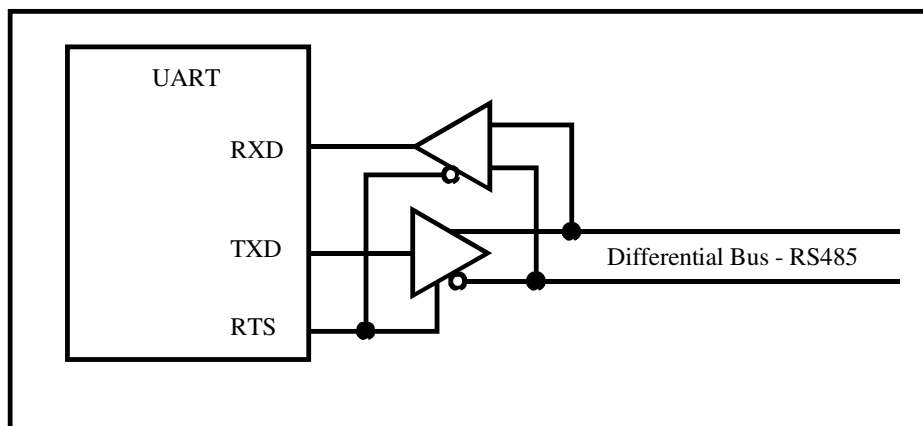


Figure 5 Standard RS485 half-duplex bus connection (RTS polarity may be configurable)

Figure 5 shows a typical circuit for driving a half-duplex RS485 transceiver. This assumes TX driving when RTS is high.

RS485 mode is not used together with flow control and the RTS signal is thus always available for transceiver control use.

Code 9 shows a method of controlling the RTS signal together with processors which do not have a HW setting to automatically control it (see appendix A for details about target processor capabilities). In order to use this method the following defines must be active, but the serial interface must not be configured for RTS/CTS flow control operation:

```
#define SUPPORT_HW_FLOW           // support RTS/CTS signals
#define UART_FRAME_COMPLETE      // the UART driver informs of frame completion
```

Furthermore, the mode setting `INFORM_ON_FRAME_TRANSMISSION` is an extended mode flag in the µTasker project which requires the configuration variable to be defined as an unsigned long in type.h:

```
typedef unsigned long UART_MODE_CONFIG; // UART mode - for extended mode support

// Initialisation code for RS485 mode
//
tInterfaceParameters.usConfig |= INFORM_ON_FRAME_TRANSMISSION;
SerialPortID = fnOpen( TYPE_TTY, FOR_I_O, &tInterfaceParameters );
...
fnDriver( SerialHandle, (MODIFY_CONTROL | CONFIG_RTS_PIN), 0);
fnDriver( SerialHandle, (MODIFY_CONTROL | SET_RTS), 0); // configure RTS pin for control use // initially drive '0'

// Message transmission code for RS485 mode
//
fnDriver( SerialHandle, (MODIFY_CONTROL | CLEAR_RTS), 0); // driver RTS '1'
fnWrite( SerialHandle, ucMessageData, length); // start message frame transmission

Code 9 - Example of configuring for RS485 mode operation and controlling the RTS
line when starting message frame transmission
```

Before use, the serial interface is configured to control the RTS output. The initial RTS state is set to '1' in the example but the polarity can be inverted if it were to needed to match the transceiver hardware.

Before starting the write of a message frame (data contained in a buffer named `ucMessage[]` with `length` in the variable `length`) the RTS line is activated by using the `fnDriver()` function as illustrated. This immediately sets the output to the '1' state to drive the RS485 bus (the polarity can be inverted if required by using `SET_RTS`) and the `fnWrite()` function caused the data transmission to commence.

When the define `UART_FRAME_COMPLETE` is enabled the user must deliver a function conforming to the prototype:

```
extern void fnUARTFrameTermination(QueueHandle Channel);
```

This function will be called by the UART driver when the last remaining data byte has been passed to the UART for transmission and can be used by the application to control the negation of the RTS line as shown in the example code 10.

```
// The UART is informing that it has just completed transmission of the last
// character in a frame.
// This is used to control RTS de-assertion, usually after a delay
//
extern void fnUARTFrameTermination(QueueHandle Channel)
{
    if (Channel == 2) {
        //fnDriver(SerialHandle, (MODIFY_CONTROL | SET_RTS), 0);
        // can only be used for certain processor types
        fnConfigureInterrupt(&timer_setup_RTS_negate);
        // start delay to RTS negation
    }
}
```

Code 10 - example of controlling negation of the RTS line on message termination

Since the timing of the last transmitted byte depends on specific UART operation in the processor used, and sometimes also on the mode of operation (interrupt or DMA driven), it is not possible to specify a generic handling of the RTS negation, which is why the user code has full control over its operation. The example shows UART channel 2 being the only one expected to call this function, although handling multiple UARTs is also possible by mapping the channel to the specific UART interface handle.

In some cases it is possible to simply negate the RTS signal using the `fnDriver()` function but it is often necessary to start a timer delay, where the resulting interrupt is used for call this action. Code 10 shows the `fnConfigureInterrupt()` function being used to configure the time delay which has been set up in the struct `timer_setup_RTS_negate`. This interface can be used with all processors, where there are however some parameters in the setup struct which may also be hardware or timer type dependent. For completeness one typical example of the set up of a hardware timer is illustrated in code 11 (*the exact configuration may vary slightly depending on the processor used*). This set up is for 19'600 Baud timing based on the fact that the specific chip type generates its final interrupt in the transmission frame (with one parity bit) 11.9 bit periods before the end of the final character.

```
static TIMER_INTERRUPT_SETUP timer_setup_RTS_negate;

timer_setup_RTS_negate.int_type = TIMER_INTERRUPT
timer_setup_RTS_negate.timer_reference = 2;
timer_setup_RTS_negate.int_priority = 3;
timer_setup_RTS_negate.int_handler = fnTimer_RTS;
timer_setup_RTS_negate.timer_mode = (TIMER_SINGLE_SHOT | TIMER_US_VALUE);
timer_setup_RTS_negate.timer_value = 620; // 620us delay before RTS negation
```

Code 11 - Example of RTS timer configuration

The important points to note are that the routine `fnTimer_RTS()` has been defined to handle the timer interrupt, where the RTS negation will actually take place (calling `fnDriver(SerialHandle, (MODIFY_CONTROL | SET_RTS), 0);`), and the delay value of 620us defines the exact time where the RTS line needs to be negated at the end of the UART transmission referenced to the `fnUARTFrameTermination()` execution for the specific hardware used. *This may also require measurements to determine the optimum setting.*

Some chips support a hardware controlled RTS mode which doesn't need software intervention. See appendix A for further details concerning the hardware target being used.

In the case of devices with HW control of RS485 mode the configuration is simplified as shown in code 12:

```
// Initialisation code for RS485 mode
//
fnDriver(SerialHandle, (MODIFY_CONTROL | CONFIG_RTS_PIN | SET_RS485_MODE), 0);
// configure RTS pin for control use
```

Code 12 - Configuring for HW RS485 mode operation

No SW control of the RTS line is required in this case since the device automatically asserts (generally sets '1') when a message frame is started and automatically negates (generally sets '0') when the stop bit of the final character has been sent.

## 9. UART as Debug Output

A UART is often used as a debug output and this is very easy to configure in the µTasker project.

The system debug output is defined by the global variable `DebugHandle`. This defaults to the value `NETWORK_HANDLE`, which means that any writes to the debug output (eg. using `fnDebugMsg("Hello, World!!");` will be passed to the user defined `fnNetworkTx()` function. This function is in `debug.c` in the µTasker demo project and will pass the output to a Telnet connection, if established.

To switch the output to an open UART interface the following assign is adequate.

```
DebugHandle = SerialPortID;           // assign our serial interface as debug port
```

`SerialPortID` is the handle of any open serial interface.

If debug output needs to be sent to more than one output, the `NETWORK_HANDLE` can be retained and the handling routine modified as required, for example:

```
extern QUEUE_TRANSFER fnNetworkTx(unsigned char *output_buffer, QUEUE_TRANSFER
nr_of_bytes)
{
    fnWrite(SerialPortID, output_buffer, nr_of_bytes);
    fnWrite(SerialPortID2, output_buffer, nr_of_bytes);
    fnWrite(USB_handle, output_buffer, nr_of_bytes);
    return (fnSendBufTCP(Telnet_socket, output_buffer, nr_of_bytes, (TCP_BUF_SEND |
TCP_BUF_SEND_REPORT_COPY)));
}
```

Code 13 – Example of sending debug output to multiple interfaces

The example Code 13 sends debug output to two serial interfaces (with handles `SerialPortID` and `SerialPortID2`, a USB port (assuming available) with handle `USB_handle` and also to a Telnet connection (if connected).

The valid debug output can be changed at any time by writing another handle to `DebugHandle`, or setting it back to the default `NETWORK_HANDLE`.

## 10. DMA Operation

When the target processor supports DMA operation on its UARTs the driver support can be activated by setting the define `SERIAL_SUPPORT_DMA` in `app_hw_xxxx.h`. This adds the DMA driver code but doesn't activate the DMA operation on the UART.

DMA operation is configured on a per UART basis when the interface is configured.

To allow a UART to operate in interrupt driven mode, the DMA operation is disabled with

```
tInterfaceParameters.ucDMAConfig = 0; // disable DMA
```

To enable DMA driven transmission on a particular UART the following is used:

```
tInterfaceParameters.ucDMAConfig = UART_TX_DMA; // activate DMA on transmission
```

DMA UART transmission operation can be used together with both XON/XOFF and RTS/CTS flow control and thus has no practical restrictions.

DMA reception operation is more limited in use due to the fact that the processor is not involved with individual characters as they are received. This means that XON/XOFF operation is generally not possible since the receiver will not recognise the control characters until a complete block of data has been received. RTS/CTS flow control is however fully operational.

The DMA reception needs to be configured to receive a block of data and the reception is only complete once the defined amount of reception bytes has been received. This restricts its use of normal user entry since the input size is generally not known in advance. Polling the DMA controller to see whether data is available would cancel any advantage of DMA reception (the fact that the data can be placed into memory without any CPU intervention) and so standard interrupt reception is preferred in this case.

The following DMA reception modes are provided for use in cases where the reception block size is known (*certain protocols with fixed length frames are suitable*) or when break control can be used to signal the completion of a reception block.

```
tInterfaceParameters.ucDMAConfig = (UART_RX_DMA | UART_RX_DMA_HALF_BUFFER);
```

The owner task will be woken when half of the input buffer is full. By setting the input buffer size to twice the standard reception frame length allows up to two reception frames to be saved by DMA reception operation, allowing the owner task to process the first data frame whilst the second is being received.

```
tInterfaceParameters.ucDMAConfig =  
    (UART_RX_DMA | UART_RX_DMA_HALF_BUFFER | UART_RX_DMA_BREAK);
```

The owner task is woken when the receiver has received a break condition after receiving a block of data. The half-buffer size should be chosen to be adequate for the largest expected frame, whereas the task would also be woken if the frame size were to be as large as the half the available buffer space. See the following section for further details of break mode support.

```
tInterfaceParameters.ucDMAConfig = (UART_RX_DMA);
```

Using `UART_RX_DMA` alone results in the receiver being woken exclusively on complete receive buffer full status. Generally the operation with wake up on half buffer full is preferred since it allows multiple message reception without critical read timing, to ensure that the buffer is emptied before a following message can be accepted.

See appendix A for details of target processors which support DMA operation on their UARTs.

## 11. Break Condition

This document is in progress. In the meantime please use the following thread for a discussion of UART use in the µTasker project:

<http://www.utasker.com/forum/index.php?topic=54.0>

## 12. Message Mode

This document is in progress. In the meantime please use the following thread for a discussion of UART use in the µTasker project:

<http://www.utasker.com/forum/index.php?topic=54.0>

## 13. Echo Mode

This document is in progress. In the meantime please use the following thread for a discussion of UART use in the µTasker project:

<http://www.utasker.com/forum/index.php?topic=54.0>

## 14. Escape Operation

This document is in progress. In the meantime please use the following thread for a discussion of UART use in the μTasker project:

<http://www.utasker.com/forum/index.php?topic=54.0>

## 15. Inter-Space Timing

This document is in progress. In the meantime please use the following thread for a discussion of UART use in the μTasker project:

<http://www.utasker.com/forum/index.php?topic=54.0>

## 16. Multi-Drop, 9-Bit Mode

9-bit mode simply means that the number of bits in the data are 9 (rather than the more typical 8 bits). This mode is not usually used to transmit 9 bits of data since this is an unusual length when the content is to be handled as bytes but rather to transmit the more normal 8 bit bytes *plus* a control information; the 9<sup>th</sup> bit of data is also sometimes referred to as the control bit.

In some situations the operation in 9-bit mode is also referred to as Multi-Drop mode of operation since the 9<sup>th</sup> bit is used to indicate which one of multiple receivers the data is being sent to – to be more precise, the receiver is first addressed by sending its identification number (or address) in a 9-bit packet with the control bit indicating that the content is an address/control and not data.

How the 9<sup>th</sup> control bit is actually used may change between systems and applications but the fact is that the UART needs to be able to send and receive 9 bits of data. Some UARTs have hardware support to do this and others don't, although they can sometimes (possibly with restrictions) be used in 8-bit mode and handle the 9<sup>th</sup> data bit via a (manipulated) parity bit.

Since the data unit is generically 9-bits the Multi-Drop mode of operation works with 16-bit units (`unsigned short`) rather than 8 bit units (`unsigned char`). This means that the user can work directly with 9 bits of data by writing the unit with the complete 9 bit content, whereby the highest 7 bits are not used.

Consider the application code wanting to send two 9 bits transfers the following code can be used:

```
unsigned short usData_9_bit[2] = {
    0x0166,
    0x0037
};
fnWrite(serial_9_bit_ID, (unsigned char *)usData_9_bit, sizeof(usData_9_bit));
```

In this example, the first 9 bit data could be addressing a device with identity 0x66 (the 9<sup>th</sup> bit, or control bit, is sent to '1' to indicate that the content is an address). The second byte is then data (control bit is '0') containing the byte 0x37.

In a similar fashion, a receiver application could read the reception data from its input buffer using:

```
unsigned short usData_9_bit[2];
fnRead(serial_9_bit_ID, (unsigned char *)usData_9_bit, sizeof(usData_9_bit));
```

The data received would be 0x0166, 0x0037 and the application can decide whether it is being addressed and then use the data if this is the case.

To be able to use the Multi-Drop mode the project define `UART_EXTENDED_MODE` must be active (this extends the configuration options to include the following). Transmission and reception support can be individually enabled by the project defines `SERIAL_MULTIDROP_TX` and `SERIAL_MULTIDROP_RX` respectively.

When the serial interface is configured, the mode is set by the configuration flags as follows:

```
tInterfaceParameters.Config |= (MULTIDROP_MODE_RX | MULTIDROP_MODE_TX);
SerialPort_9BIT_ID = fnOpen(TYPE_TTY, ucDriverMode, &tInterfaceParameters);
```

In this example the UART interface is opened with both receiver and transmitter operating in 9-bit mode. *In some cases the UART HW may not only be able to operate in the same mode in both directions.*

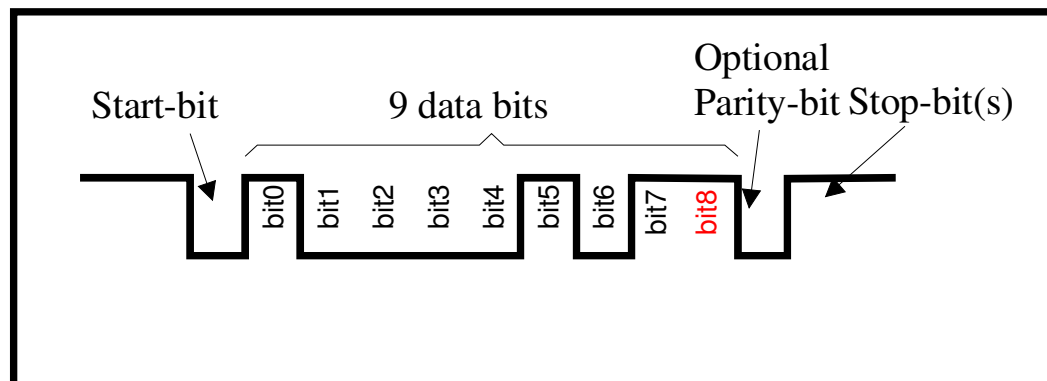
To note are the following points:

- 1) The size of the input and output buffers are in bytes. A 9 bit transmission or reception unit occupies two byte spaces in the buffer. The buffer size may therefore need to be dimensioned larger in comparison to 8 bit mode due to this fact.
- 2) The application can work directly with unsigned short arrays, whereby the 9 bits or transmitted or received data occupy the 9<sup>th</sup> least significant bits of each short word. The control bit is the 9<sup>th</sup> bit (bit 8) and the 8 bytes of data that it carries are in the least significant byte (bits 0..7).
- 3) When sending or receiving 9-bit data each unit occupies 2 bytes in the UART buffer, therefore the length written or received needs to be the total size (2 for each 9-bit unit) and not 1 for each unit!
- 4) The translation between the 16 bit application units and the 8 bit buffer is performed automatically in the UART driver. The driver may perform some translation to respect the little-endian or big-endian operation of the processor in order to keep the interface compatible.
- 5) Application code will probably work with a control bit defines such as
 

```
#define DATA_CONTROL 0x0100
#define DATA_CONTENT 0x0000
```

 When sending the code could thus look like
 

```
unsigned short usData_9_bit[2] = {(DATA_CONTROL | 0x66) , 0x37};
```
- 6) The location of the 9th data bit in the UART transmission is illustrated in the following diagram:



## 17. Changing Operation Modes and Flushing Queues

This document is in progress. In the meantime please use the following thread for a discussion of UART use in the μTasker project:

<http://www.utasker.com/forum/index.php?topic=54.0>

## 18. UARTs in the μTasker Simulator

This document is in progress. In the meantime please use the following thread for a discussion of UART use in the μTasker project:

<http://www.utasker.com/forum/index.php?topic=54.0>

## 19. External UARTs

Most processors offer at least one UART as standard on-chip peripheral. In fact the number is more often 2 or 3, and some devices, such as the Kinetis K60 have even 5 on-chip ones. However, even though the COM port is gradually disappearing on the modern PC this interface remains so popular in control applications that sometimes even all internal UARTs don't suffice for a particular application and this is where external UARTs come to the rescue.

External UARTs can be memory mapped into the system when the processor has an external memory bus but often single-chip processors in small packages use SPI or I2C buses to connect to external UART devices.

The μTasker project supports the **NXP SC16IS762/52** via SPI interface as dual external UART expansion device. A single SC16IS752 (the 62 version can achieve faster communication speeds) has two UART channels and multiple devices can be connected, each with its own chip select line, to extend the UART quantity where necessary.

In order to activate external UART channels, the define `NUMBER_EXTERNAL_SERIAL` (in `app_hw_XXXX.h`) needs to be increased from 0 to the number required; 2 for each SC16IS752 added. In addition, the define `EXT_UART_SC16IS7XX` should be active so that the device's driver is also included.

The numbering of the UART channels starts with the internal ones. If, for example, a processor has 3 internal UARTs these are numbered 0, 1 and 2. If a higher value is specified and external UARTs are available the external ones will be used; in the case of 2 external SC16IS752s these will have the channel numbers 3, 4 (in first chip), 5 and 6 (in the second chip). The μTasker TTY driver is compatible with all UART types and so the user doesn't generally need to know a great deal about their details. The processor's UART driver will pass on the handling of all non-internal UARTs to the external device driver. In the case of the SAM7X, for example, this external driver is contained in the file `spi_sc16IS7xx.h`, which is included in `SAM7X.c`.

## 20. Conclusion

### Modifications:

V0.0 5.4.2009:

- Initial draft – work in progress. Not officially released.

V0.01 11.4.2009:

- Initial draft – work in progress. Reception examples and flow control test configuration added.

V0.02 11.4.2009:

- Initial draft – work in progress. Transmission, transmitter blocking and RS485 mode added, plus RTS timings in appendix A

V0.03 19.4.2009:

- Initial draft – work in progress. UART debug output, DMA modes.

V0.04 11.05.2011:

- Initial draft – work in progress. External UART section and 9-bit (multi-drop) mode added.

## Appendix A – Hardware Dependencies

### a) UART Peripheral Pins

TO DO. Add target specific peripheral configuration details

### b) RS485 mode

The following results were obtained by measuring the final transmission interrupt of the UART interface, operating at 19200Baud with parity enabled. *Devices supporting DMA operation were also measured in DMA mode, where the final interrupt corresponds to the DMA completion interrupt.*

The values can be used to configure a timer (where required) to generate a time delay suitable for deactivating the RTS signal as the final stop bit is sent in a message frame.

#### *Freescale Coldfire V2 MCU*

- M522xx Interrupt 19'200 Baud (Bit Period 52us). RTS activated about 17us before first bit. Last character interrupt about 10.9 bit times before end (0.57ms).
- M522xx DMA 19'200 Baud (Bit Period 52us). RTS activated about 17us before first bit. Last character (DMA complete) interrupt about 22 bit times before end (1.144ms).

#### *ATMEL SAM7X*

- SAM7X Interrupt 19'200 Baud (Bit Period 52us). RTS activated about 20us..70us (varies due to synchronisation to the UART clock) before first bit. Last character interrupt about 9.8 bit times before end (0.512ms).  
Note: UART2 (using ports as RTS) has activation stable at about 37us and identical off delay.
- SAM7X DMA 19'200 Baud (Bit Period 52us) [UART0 and UART1]. RTS activated about 20us..70us (varies due to synchronisation to the UART clock) before first bit. Last character (DMA complete) interrupt about 21 bit times before end (1.08ms).  
Note: UART2 (using ports as RTS) has activation stable at about 37us and identical off delay.
- SAM7X in RS485 mode 19'200 Baud (Bit Period 52us) [UART0 and UART1 only]. RTS activated about 20us..35us (varies due to synchronisation to the UART clock) before first bit. RTS is active High. RTS automatically negated (low) exactly at the end of the stop bit. (*By setting US\_TTGR to 1..255 additional bit period delays can be added*).

*Luminary Micro*

- LM3Sxxxx Interrupt 19'200 Baud (Bit Period 52us). RTS activated about 13us before first bit. Last character interrupt about 11.9 bit times before end (0.62ms).
- LM3Sxxxx DMA 19'200 Baud (Bit Period 52us). RTS activated about 13us before first bit. DMA transmission complete interrupt about 11.9 bit times before end (0.62ms).

*NXP LPC23XX*

- LPC23XX Interrupt 19'200 Baud (Bit Period 52us). RTS activated about 20us..70us (varies due to synchronisation to the UART clock) before first bit. Last character interrupt about 1 bit before end (43us).