Using UART in CC111xFx, CC243xFx, CC251xFx and CC253xFx

By Torgeir Sundet

Keywords

- *UART*
- *CPU*
- *DMA*
- *Protocol*
- *Baud Rate*
- *Asynchronous*
- *CC1110Fx*
- *CC1111Fx*
- *CC2430Fx*
- *CC2431Fx*
- *CC2510Fx*
- *CC2511Fx*
- *CC2530Fx*
- *CC2531Fx*

1 Introduction

This design note describes the key elements and simple usage of the CC111xFx/CC243xFx/CC251xFx/CC253xFx USART peripheral in UART mode. The main objective is to explain the CC111xFx/CC243xFx/CC251xFx/CC253xFx software required to operate the UART, with and without DMA support.

The UART implements asynchronous serial communication, and is typically used as a diagnostic/command interface. In the following sections an x represents USART number 0 or 1 if nothing else is stated.

Table of Contents

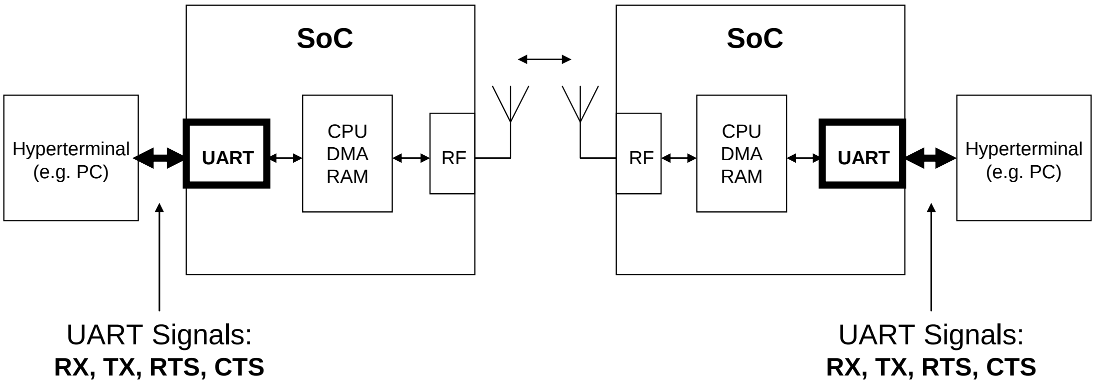
2 Abbreviations

EB	Evaluation Board
CPU	Central Processing Unit
CTS	Clear To Send
DMA	Direct Memory Access
DCE	Data Circuit-terminating Equipment, according to [5].
DTE	Data Terminal Equipment, according to [5].
GPIO	General Purpose Input and Output
HS XOSC	High Speed Xtal Oscillator, refer to relevant data sheets [1], [2], [3] and [4] for actual frequency.
HW	Hard Ware
ISR	Interrupt Service Routine
NA	Not Applicable
NOP	No Operation
SmartRF®	Registered Trademark, used in this document to reference the Low Power RF product line from Texas Instruments.
SoC	System on Chip. A collective term used to refer to Texas Instruments ICs with on-chip MCU and RF transceiver. Used in this document to reference the CC1110, CC1111, CC2430, CC2431, CC2510, CC2511, CC2530 and CC2531.
RF	Radio Frequency.
RTS	Ready To Send (implies Ready to Receive for DTE to DTE communication)
RX	Receive. Used in this document to reference UART/Radio receive.
RS232	Recommended Standard 232, a standard defining the format and signal levels of a specific asynchronous serial interface.
TTL	Transistor-transistor Logic
TX	Transmit. Used in this document to reference UART/Radio transmit.
UART	Universal Asynchronous serial Receiver and Transmitter
UART protocol	Used in this document to represent the UART signalling format; start/stop bit, data bit, parity.
USART	Universal Synchronous/Asynchronous serial Receiver and Transmitter.

3 Scope

In UART mode the SoC USART peripheral can be used to interface any external device (assuming TTL voltage level) which supports the UART protocol, meaning half/full-duplex asynchronous serial transfer of 8 bit data. A common application would use the UART to support a wireless modem, as shown in Figure 1. However, this design note focuses on the fundamental UART receive/transmit operation, and how to use the different protocol settings, including start/stop bit and parity.

Two main UART transmit/receive methods are described; using the UART with DMA support, and using it without DMA support. Furthermore, when using the UART without DMA support, two separate implementations are covered; UART transmit/receive by polling the UART Interrupt Request Flags, and transmit/receive using UART ISR.



Key UART Features:
- Designated SoC peripheral
- Wired Asynchronous Serial Bi-directional Communication
- Frame Recognition via Start/Stop bit
- Data Level Recognition via Over-sampling => No Clock
- Optional Data Flow Control - Hardware handshaking
- Optional Data Integrity Control – Parity Check

Figure 1: Typical Application with UART Support

4 Using the USART Peripheral in UART Mode

The UART peripheral uses `UxCSR.MODE` to determine the desired USART operation mode (UART or SPI) and the `UxUCR/UxGCR` registers to configure the UART format/signaling; start/stop bit, data bit and parity. For data transfer on the UART interface the `UxDBUF` registers are used. Internal data transfer, between the SoC memory and UART can be done by the CPU or the DMA controller.

Using the CPU for internal data transfer prevents the CPU from performing other tasks (except from ISRs) during memory transfer. Using the DMA controller allows the CPU to continue with other tasks, while the DMA controller transfers data between the UART and SoC memory. In a UART application it is typically desired that the DMA controller handles block transfers, not just transfers of a single byte. This means that the DMA controller must be configured to interface allocated RX/TX buffers and either download data (transmit) from the allocated TX buffer to the `UxDBUF`, or upload data (receive) from the `UxDBUF` to the allocated RX buffer. A DMA ISR can typically be implemented to automatically start a new UART transmit/receive session upon completing each block transfer. This would be relevant when the application needs efficient streaming of data between e.g. RF and UART.

4.1 Mapping the USART Peripheral in UART Mode

In order for the UART to be mapped to the desired pins and generate the associated interrupt requests and DMA triggers, the following register/descriptor fields must be set according to the desired configuration/functionality (for cross referencing, please see Figure 2):

- `PxSEL.SELPx_y` (see Figure 2 for correlation of x and y)
  Selects whether Port x Pin y shall be GPIO or mapped to the UART.

- `IEN2.UTXxIE`
  UART TX CPU interrupt enable/disable.

- `IEN0.URXxIE`
  UART RX CPU interrupt enable/disable.

- `IEN0.EA`
  Global/master interrupt enable/disable.

- `IEN1.DMAIE`
  DMA CPU interrupt enable/disable.

- `DMA Descriptor.IRQMASK`
  DMA Transfer Complete interrupt enable/disable.

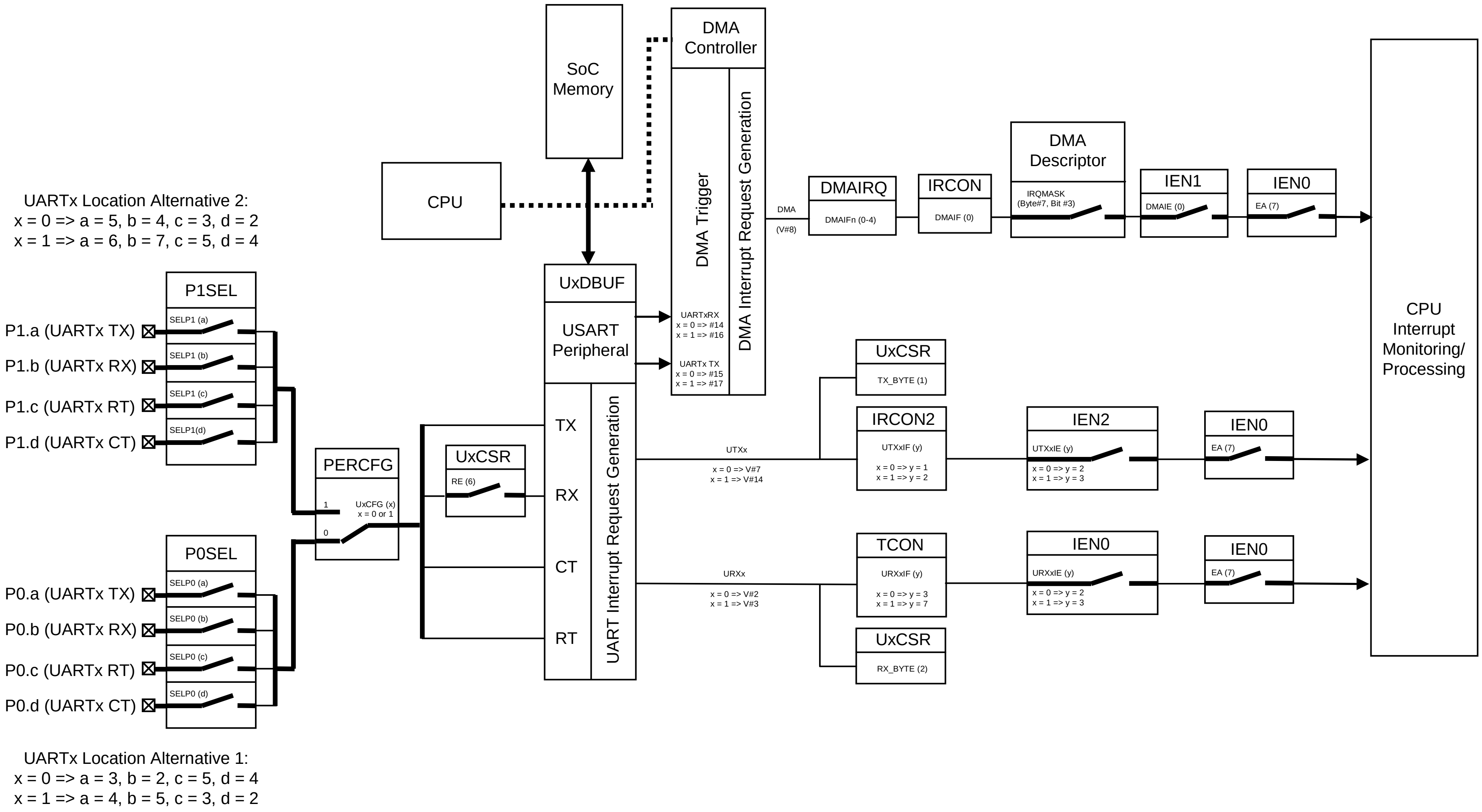
The corresponding UART / DMA interrupt flags are located in the register fields listed below. If the corresponding interrupt enable flags have been set, then an interrupt request causes the CPU to vector its code execution to the associated UART / DMA ISR (for cross referencing, please see Figure 2):

- `UxCSR.TX_BYTE` and `IRCON2.UTXxIF`
  Indicates that the UART has transmitted a byte, and that it is ready to transmit another byte.

- `UxCSR.RX_BYTE` and `TCON.URXxIF`
  Indicates that the UART has received a byte, and that it is ready to receive another byte.

- `IRCON.DMAIF` and `DMAIRQ.DMAIFn`
  Indicates that the DMA controller has reached its transfer count; that is; it has transferred a defined range of data between the UART and SoC memory. As generally noted above, if the corresponding DMA interrupt enable flags have been set, the CPU would automatically vector its code execution to a DMA ISR, and thus allow the CPU to prepare another block transfer between the UART and SoC memory.

If supported by the DMA controller in UART TX then DMA trigger #15/17 will initiate a single byte DMA transfer from the allocated UART TX source buffer to the `UxDBUF` register. For UART RX DMA trigger #14/16 will initiate a single byte DMA transfer from the `UxDBUF` register to the allocated UART RX destination buffer.



UARTx Location Alternative 2:
x = 0 => a = 5, b = 4, c = 3, d = 2
x = 1 => a = 6, b = 7, c = 5, d = 4

UARTx Location Alternative 1:
x = 0 => a = 3, b = 2, c = 5, d = 4
x = 1 => a = 4, b = 5, c = 3, d = 2

Figure 2: Mapping the USART Peripheral in UART Mode

4.1.1 Mapping the UART to the SoC I/O

With reference to Figure 2, the SoC I/O map shown in Table 1 (extract from the “Peripheral I/O” section in the SoC data sheets ([1], [2], and [3]) applies for mapping the UART peripheral to the SoC I/O port:

Periphery / Function		P0								P1								P2				
		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	4	3	2	1	0
USART0	Alt 1			RT	CT	TX	RX															
	Alt 2											TX	RX	RT	CT							
UART1	Alt. 1			RX	TX	RT	CT															
	Alt 2									RX	TX	RT	CT									

Table 1: SoC I/O Map for the UART

The required code is shown in Figure 3, and implements the following main steps:

1. Set `PERCFG.UxCFG` such that the UART connects to the relevant target SoC port pins.

2. Set `PxSEL.SELPx_y` such that the allocated SoC port pins are configured for UART functionality (see Figure 2 for correlation of x and y).

```
// C language code:

// This function maps/connects the UART to the desired SoC I/O port.
// The application should call this function with "uartPortAlt" = 1 or 2,
// and "uartNum" = 0 or 1.

void uartMapPort(unsigned char uartPortAlt, unsigned char uartNum) {

  // If UART Port Alternative 1 desired
  if(uartPortAlt == 1) {
    // If UART0 desired
    if (uartNum == 0) {
      // Configure UART0 for Alternative 1 => Port P0 (PERCFG.U0CFG = 0)
      PERCFG &= ~0x01;
      // Configure relevant Port P0 pins for peripheral function:
      // P0SEL.SELP0_2/3/4/5 = 1 => RX = P0_2, TX = P0_3, CT = P0_4, RT = P0_5
      P0SEL |= 0x3C;
      // Configure relevant Port P1 pins back to GPIO function
      P1SEL &= ~0x3C;
    // Else (UART1 desired)
    } else {
      // Configure UART1 for Alternative 1 => Port P0 (PERCFG.U1CFG = 0)
      PERCFG &= ~0x02;
      // Configure relevant Port P0 pins for peripheral function:
      // P0SEL.SELP0_2/3/4/5 = 1 => CT = P0_2, RT = P0_3, TX = P0_4, RX = P0_5
      P0SEL |= 0x3C;
      // Configure relevant Port P1 pins back to GPIO function
      P1SEL &= ~0xF0;
    }
  // Else (UART Port Alternative 2 desired)
  } else {
    // If UART0 desired
    if (uartNum == 0) {
      // Configure UART0 for Alternative 2 => Port P1 (PERCFG.U0CFG = 1)
      PERCFG |= 0x01;
      // P1SEL.SELP1_2/3/4/5 = 1 => CT = P1_2, RT = P1_3, RX = P1_4, TX = P1_5
      P1SEL |= 0x3C;
      // Configure relevant Port P0 pins back to GPIO function
      P0SEL &= ~0x3C;
    // Else (UART1 desired)
    } else {
      // Configure UART1 for Alternative 2 => Port P1 (PERCFG.U1CFG = 1)
      PERCFG |= 0x02;
      // P1SEL.SELP1_4/5/6/7 = 1 => CT = P1_4, RT = P1_5, TX = P1_6, RX = P1_7
      P1SEL |= 0xF0;
      // Configure relevant Port P0 pins back to GPIO function
      P0SEL &= ~0x3C;
    }
  }

}
```

Figure 3: Mapping the UART to the SoC I/O

4.1.2 Interfacing the UART with the SmartRF®04EB

When interfacing the UART with the SmartRF®04EB there are two possible configurations, that is; SoC to SoC, or SoC to PC. Figure 4 and Figure 5 shows the implemented wiring for these configurations, please refer Table 1 for the internal SoC pin mapping. Note that the SoC is implemented as a so-called DTE device. This means that the SoC RT (Ready-To-Send or Ready-To-Receive) is an output signal to control the data input flow, while the CT signal is a SoC input, gating the SoC data output flow. Setting up a SoC to SoC connection consequently requires a null-modem (cross over) connection (Figure 4). Interfacing a PC RS232 port requires a straight connection (Figure 5). For more detailed explanation of the fundamental UART/RS232 terminology, signaling/format and cable interface, please refer to [5], [6], and [7].

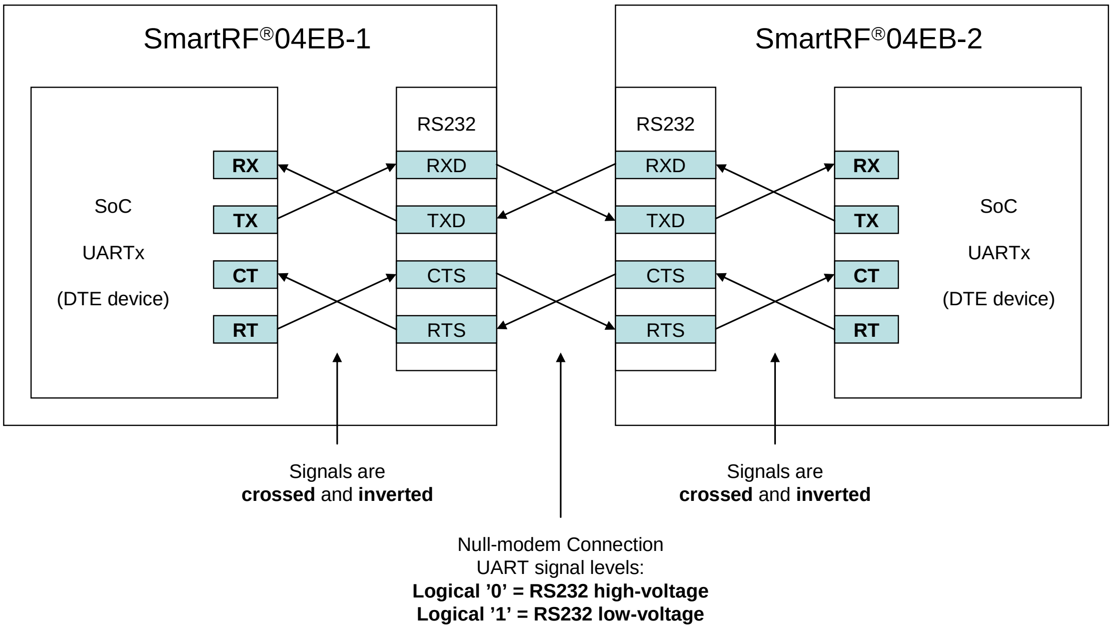


Figure 4: Interfacing UART with SmartRF®04EB - SoC to SoC

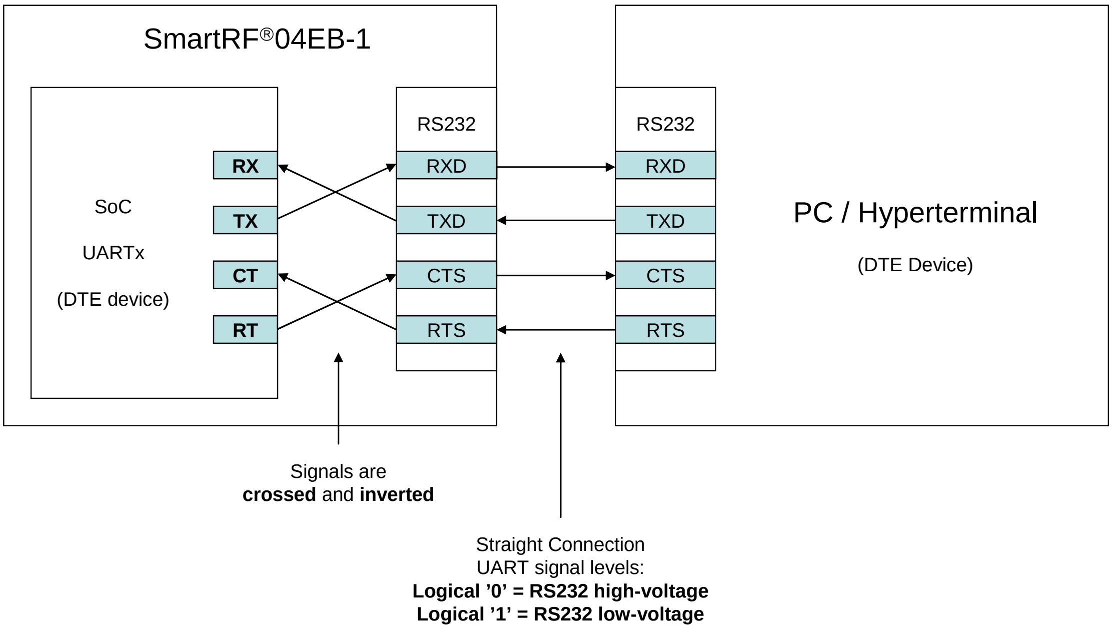


Figure 5: Interfacing UART with SmartRF®04EB - SoC to PC

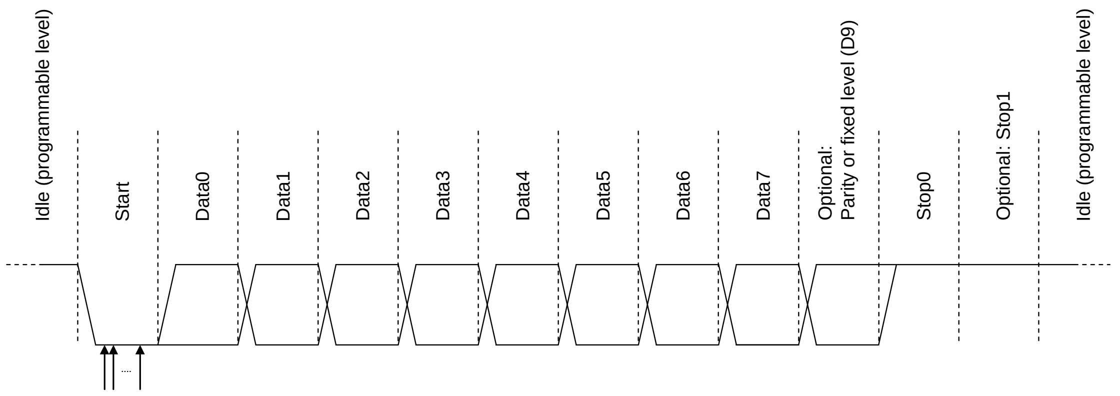
4.1.3 Setting up the UART Protocol

4.1.3.1 UART Frame and Flow

The UART interface consists of 4 signals:

- Serial Data In (RX)
- Serial Data Out (TX)
- Ready To Send (RT/RTS)
- Clear To Send (CT/CTS)

The UART data is communicated on the RX/TX lines according to the format shown in Figure 6. Designated start and stop bits are used to identify/recognize each basic UART packet/frame, which holds 8 data bit. In 9 bit mode (`UxUCR.BIT9 = 1`) a designated parity bit is used for data integrity/error checking. Alternatively the parity bit can be replaced by a 9th bit fixed-level according to the value of `UxUCR.D9`. Note that a UART protocol recovers the bit clock based on the pre-programmed baudrate (`UxBAUD.BAUD_M` and `UxGCR.BAUD_E`) and built-in over-sampling capability. Consequently, the UART does not use a designated clock signal for this purpose.



Level decision at UART receiver:
• Over-samples RX line to determine level
• No separate clock line

Figure 6: UART Frame Format

In order to prevent data overflow the UART offers built-in HW flow control, using the RT/RTS and CT/CTS signals. When an external UART device de-asserts (TTL high) the CT/CTS line then the SoC UART halts its current TX operation until the external UART device asserts (TTL low) CT/CTS again (ready to receive new data). In RX mode the SoC UART prevents data overflow by automatically de-asserting (TTL high) the RT/RTS signal (provided that this signal is connected to CT/CTS on the external UART device). This occurs if both `UxDBUF` and the internal UART RX shift register are full. The UART SoC will assert (TTL low) the RT/RTS signal again (allowing data input flow) once the internal UART RX shift register is emptied, that is; latched over to `UxDBUF` as a result of application-read of `UxDBUF`.

Note: when the application has read `UxDBUF` *it is very important that it does not clear* `UxCSR.RX_BYTE`*. Clearing* `UxCSR.RX_BYTE` *implicitly makes the UART believe that the UART RX shift register is empty, even though it might hold pending data (typically due to back-to-back transmission). Consequently the UART asserts (TTL low) the RT/RTS line, which allows flow into the UART, leading to potential overflow. Hence the* `UxCSR.RX_BYTE` *flag integrates closely with the automatic RT/RTS function and must therefore be controlled solely by the SoC UART it self. Otherwise the application could typically experience that the RT/RTS line remains asserted (TTL low) even though a back-to-back transmission clearly suggests it ought to intermittently pause the flow.*

A simplified view of the UART RX flow is shown in Figure 7 and Figure 8. In RX mode the UART over-samples the RX line to determine the bit level. Each data bit is then shifted into the internal shift register, which is latched to UxDBUF after the 8th data bit. Finally the received data byte is read, typically into an allocated target buffer. In 9 bit mode (Figure 8) the UART will monitor the 9th bit according to the selected Parity mode (Odd or Even). Note that, if HW flow control is enabled, then the UART will automatically control the RT/RTS signal based on the pending-status of the internal UART RX shift register.

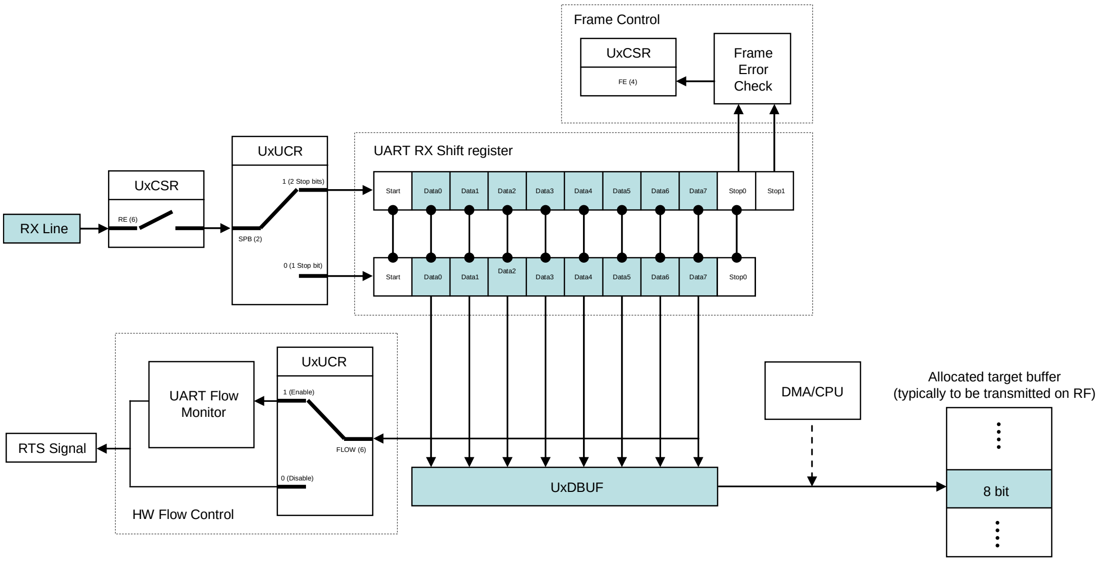


Figure 7: UART RX Flow in 8 bit mode, no Parity Control, optional HW Flow Control

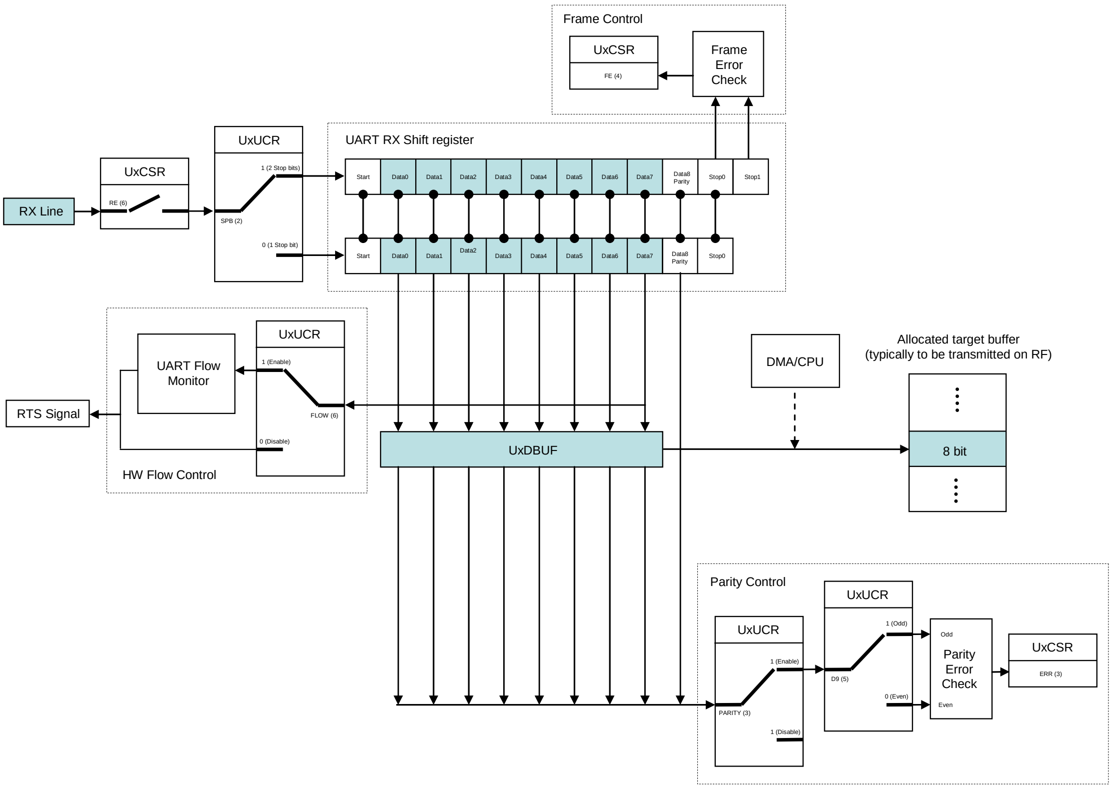


Figure 8: UART RX Flow in 9 bit mode, optional Parity Control, optional HW Flow Control

A simplified view of the UART TX flow is shown in Figure 9 and Figure 10. In TX mode the data to be transmitted on UART is typically located in an allocated source buffer. Each data byte is written to the UxDBUF, which then triggers the UART to load its internal shift register and finally output the data byte serially on the TX line. In 9 bit mode (Figure 10) the UART will automatically set the 9th bit to a fixed (programmable) level or according to the selected Parity mode (Odd or Even). Note that, if HW flow control is enabled, then the UART will not load the shift register as long as the associated CT/CTS pin (SoC input) is de-asserted (TTL high).

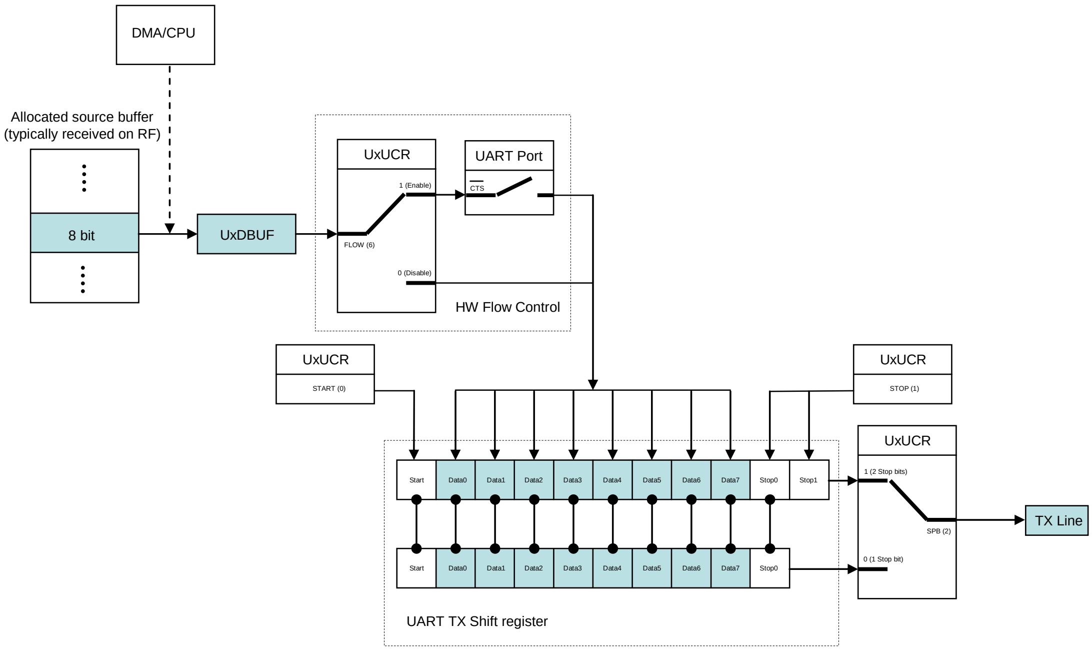


Figure 9: UART TX Flow in 8 bit mode, no Parity, optional HW Flow Control

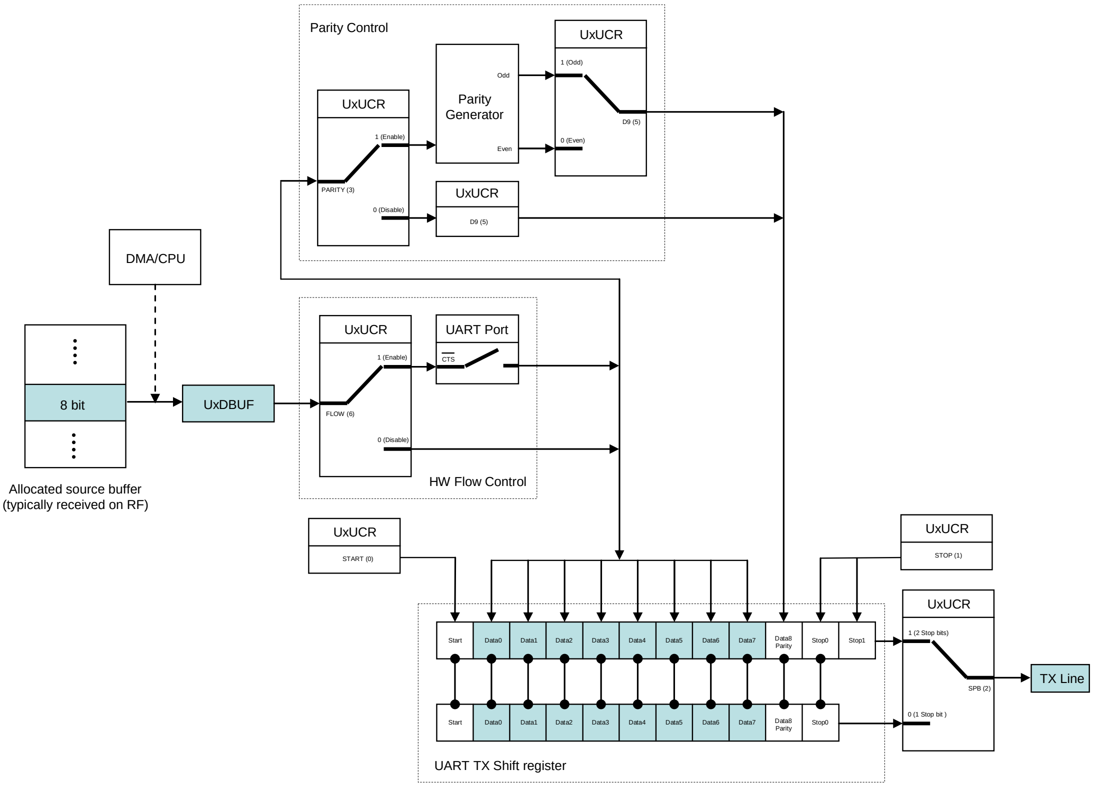


Figure 10: UART TX Flow in 9 bit mode, optional Parity Control, optional HW Flow Control

4.1.3.2 UART Baud Rate Generation/Initialization

The SoC uses an internal baud rate generator to implement the desired data bit rate, and initialization is done according to the formula shown in Equation 1:

$$Baudrate = \frac{(256 + BAUD\_M) \cdot 2^{BAUD\_E}}{2^{28}} \cdot F$$

Equation 1: UART Baud Rate Generation

The variables in Equation 1 represent the following parameters:

- F : The System Clock Frequency
- BAUD_M : The 8 bit Baud rate Mantissa located in `UxBAUD.BAUD_M[7:0]`
- BAUD_E : The 5 bit Baud rate Exponent located in `UxGCR.BAUD_E[4:0]`

Some typical UART Baud Rate settings are shown in Table 2, Table 3, and Table 4.

Baud Rate [bps]	UxBAUD.BAUD_M	UxGCR.BAUD_E	Error (%)
2400	131	6	0.04
9600	131	8	0.04
57600	34	11	0.13
115200	34	12	0.13
230400	34	13	0.13

Table 2: Commonly used Baud Rate Settings for 26 MHz System Clock

Baud Rate [bps]	UxBAUD.BAUD_M	UxGCR.BAUD_E	Error (%)
2400	163	6	0.08
9600	163	8	0.09
57600	59	11	0.14
115200	59	12	0.14
230400	59	13	0.14

Table 3: Commonly used Baud Rate Settings for 24 MHz System Clock

Baud rate (bps)	UxBAUD.BAUD_M	UxGCR.BAUD_E	Error (%)
2400	59	6	0.14
9600	59	8	0.14
57600	216	10	0.03
115200	216	11	0.03
230400	216	12	0.03

Table 4: Commonly used Baud Rate Settings for 32 MHz System Clock

The required code for initializing the UART Baud Rate Generator is shown in Figure 11 and implements the following main steps:

1. Switch the system clock to HS XOSC for consistent UART clock generation.

2. Set BAUD_M and BAUD_E according to the formula in Equation 1.

```
// C language code:

// This function initializes the UART bit rate.

void uartInitBitrate(unsigned char uartBaudM, unsigned char uartBaudE) {

  ///////////////////////////////////////////////////////////////
  // This initial code section ensures that the SoC system clock is driven
  // by the HS XOSC:

  // Clear CLKCON.OSC to make the SoC operate on the HS XOSC.
  // Set CLKCON.TICKSPD/CLKSPD = 000 => system clock speed = HS RCOSC speed.
  CLKCON &= 0x80;

  // Monitor CLKCON.OSC to ensure that the HS XOSC is stable and actually
  // applied as system clock source before continuing code execution
  while(CLKCON & 0x40);

  // Set SLEEP.OSC_PD to power down the HS RCOSC.
  SLEEP |= 0x04;
  ///////////////////////////////////////////////////////////////

  // Initialize bitrate (U0BAUD.BAUD_M, U0GCR.BAUD_E)
  U0BAUD = uartBaudM;
  U0GCR = (U0GCR&~0x1F) | uartBaudE;
}
```

Figure 11: Initializing the UART Baud Rate Generator

4.1.3.3 Initialization of the UART protocol (Start/Stop bit, Data bits, Parity Control, Flow Control)

The required code for initializing the UART protocol is shown in Figure 12 and Figure 13, and implements the following main steps:

1. Set `UxCSR.MODE = 1` to configure the UART for UART function.
   Note that `UxCSR.MODE = 0` implies SPI mode.

2. Set `UxUCR.START` according to desired start bit level.
   Typical: `UxUCR.START = 0` => low level.

3. Set `UxUCR.STOP` according to desired stop bit level.
   Typical: `UxUCR.STOP` = 1 => high level.

4. Set `UxUCR.SPB` according to desired number of stop bits.
   Typical: `UxUCR.SPB = 0` => 1 stop bit.

5. Set `UxUCR.PARITY` to disable/enable parity mode.
   Note: if parity is enabled then remember to also set `UxUCR.BIT9 = 1`.

6. Set `UxUCR.BIT9` according to desired number of data bits (8 or 9).

7. If parity disabled then set `UxUCR.D9` according to the desired fixed level on 9th bit.
   If parity enabled then set `UxUCR.D9` according to the desired parity mode (even/odd).

8. Set `UxGCR.ORDER` according to desired bit order (LSB or MSB first).

```
// C language code:

// Define and allocate a setup structure for the UART protocol:

typedef struct {
  unsigned char uartNum               : 1;  // UART peripheral number (0 or 1)
  unsigned char START                 : 1;  // Start bit level (low/high)
  unsigned char STOP                  : 1;  // Stop bit level (low/high)
  unsigned char SPB                   : 1;  // Stop bits (0 => 1, 1 => 2)
  unsigned char PARITY                : 1;  // Parity control (enable/disable)
  unsigned char BIT9                  : 1;  // 9 bit enable (8bit / 9bit)
  unsigned char D9                    : 1;  // 9th bit level or Parity type
  unsigned char FLOW                  : 1;  // HW Flow Control (enable/disable)
  unsigned char ORDER                 : 1;  // Data bit order(LSB/MSB first)
} UART_PROT_CONFIG;

UART_PROT_CONFIG __xdata uartProtConfig;

// Define size of allocated UART RX/TX buffer (just an example)
#define SIZE_OF_UART_RX_BUFFER   50
#define SIZE_OF_UART_TX_BUFFER   SIZE_OF_UART_RX_BUFFER

// Allocate buffer+index for UART RX/TX
unsigned short __xdata uartRxBuffer[SIZE_OF_UART_RX_BUFFER];
unsigned short __xdata uartTxBuffer[SIZE_OF_UART_TX_BUFFER];
unsigned short __xdata uartRxIndex, uartTxIndex;
```

Figure 12: Allocating UART Buffers and Protocol Setup Descriptor

```
// C language code:
// This function initializes the UART protocol (start/stop bit, data bits,
// parity, etc.). The application must call this function with an initialized
// data structure according to the code in Figure 12.

void uartInitProtocol(UART_PROT_CONFIG* uartProtConfig) {

  // Initialize UART protocol for desired UART (0 or 1)
  if (uartProtConfig->uartNum == 0) {
    // USART mode = UART (U0CSR.MODE = 1)
    U0CSR |= 0x80;
    // Start bit level = low => Idle level = high (U0UCR.START = 0)
    // Start bit level = high => Idle level = low (U0UCR.START = 1)
    U0UCR = (U0UCR&~0x01) | uartProtConfig->START;
    // Stop bit level = high (U0UCR.STOP = 1)
    // Stop bit level = low (U0UCR.STOP = 0)
    U0UCR = (U0UCR&~0x02) | (uartProtConfig->STOP << 1);
    // Number of stop bits = 1 (U0UCR.SPB = 0)
    // Number of stop bits = 2 (U0UCR.SPB = 1)
    U0UCR = (U0UCR&~0x04) | (uartProtConfig->SPB << 2);
    // Parity = disabled (U0UCR.PARITY = 0)
    // Parity = enabled (U0UCR.PARITY = 1)
    U0UCR = (U0UCR&~0x08) | (uartProtConfig->PARITY << 3);
    // 9-bit data disable = 8 bits transfer (U0UCR.BIT9 = 0)
    // 9-bit data enable = 9 bits transfer (U0UCR.BIT9 = 1)
    U0UCR = (U0UCR&~0x10) | (uartProtConfig->BIT9 << 4);
    // Level of bit 9 = 0 (U0UCR.D9 = 0), used when U0UCR.BIT9 = 1
    // Level of bit 9 = 1 (U0UCR.D9 = 1), used when U0UCR.BIT9 = 1
    // Parity = Even (U0UCR.D9 = 0), used when U0UCR.PARITY = 1
    // Parity = Odd (U0UCR.D9 = 1), used when U0UCR.PARITY = 1
    U0UCR = (U0UCR&~0x20) | (uartProtConfig->D9 << 5);
    // Flow control = disabled (U0UCR.FLOW = 0)
    // Flow control = enabled (U0UCR.FLOW = 1)
    U0UCR = (U0UCR&~0x40) | (uartProtConfig->FLOW << 6);
    // Bit order = MSB first (U0GCR.ORDER = 1)
    // Bit order = LSB first (U0GCR.ORDER = 0) => For PC/Hyperterminal
    U0GCR = (U0GCR&~0x20) | (uartProtConfig->ORDER << 5);
  } else {
    // USART mode = UART (U1CSR.MODE = 1)
    U1CSR |= 0x80;
    // Start bit level = low => Idle level = high (U1UCR.START = 0)
    // Start bit level = high => Idle level = low (U1UCR.START = 1)
    U1UCR = (U1UCR&~0x01) | uartProtConfig->START;
    // Stop bit level = high (U1UCR.STOP = 1)
    // Stop bit level = low (U1UCR.STOP = 0)
    U1UCR = (U1UCR&~0x02) | (uartProtConfig->STOP << 1);
    // Number of stop bits = 1 (U1UCR.SPB = 0)
    // Number of stop bits = 2 (U1UCR.SPB = 1)
    U1UCR = (U1UCR&~0x04) | (uartProtConfig->SPB << 2);
    // Parity = disabled (U1UCR.PARITY = 0)
    // Parity = enabled (U1UCR.PARITY = 1)
    U1UCR = (U1UCR&~0x08) | (uartProtConfig->PARITY << 3);
    // 9-bit data enable = 8 bits transfer (U1UCR.BIT9 = 0)
    // 9-bit data enable = 8 bits transfer (U1UCR.BIT9 = 1)
    U1UCR = (U1UCR&~0x10) | (uartProtConfig->BIT9 << 4);
    // Level of bit 9 = 0 (U1UCR.D9 = 0), used when U1UCR.BIT9 = 1
    // Level of bit 9 = 1 (U1UCR.D9 = 1), used when U1UCR.BIT9 = 1
    // Parity = Even (U1UCR.D9 = 0), used when U1UCR.PARITY = 1
    // Parity = Odd (U1UCR.D9 = 1), used when U1UCR.PARITY = 1
    U1UCR = (U1UCR&~0x20) | (uartProtConfig->D9 << 5);
    // Flow control = disabled (U1UCR.FLOW = 0)
    // Flow control = enabled (U1UCR.FLOW = 1)
    U1UCR = (U1UCR&~0x40) | (uartProtConfig->FLOW << 6);
    // Bit order = MSB first (U1GCR.ORDER = 1)
    // Bit order = LSB first (U1GCR.ORDER = 0) => For PC/Hyperterminal
    U1GCR = (U1GCR&~0x20) | (uartProtConfig->ORDER << 5);
  }
}
```

Figure 13: Initializing the UART Protocol

4.2 Using the UART without DMA Support

Using the UART without support from the DMA controller means that the CPU must handle all data transfers between the UART and SoC memory. During UART transfers the CPU will not be able to perform other tasks, except from ISRs. In order to process the UART communication without DMA support the application can either poll the UART Interrupt Flag, or implement a designated UART ISR. Both methods imply that the application/CPU must monitor the UART interrupt request flags; `TCON.URXxIF` and `IRCON2.UTXxIF`.

4.2.1 Processing UART Communication using UART Polling

The following section shows how to control the UART TX/RX communication, using polling of the interrupt request flags. This method typically implies that the application rests all other tasks (apart from ISRs) while processing the UART communication, that is; accessing the `UxDBUF`, and `TCON.URXxIF` and `IRCON2.UTXxIF`.

4.2.1.1 UART TX Processing using UART Polling

The required code for UART TX, with polling, is shown in Figure 14 and implements the following main steps:

1. Allocate UART TX buffer for the associated source data.

2. Disable the UART TX interrupt (this instruction will not mask the generation of the interrupt request flags, used for polling UART TX complete; ref. Figure 2).

3. Read from the allocated TX source buffer and write to the appropriate `UxDBUF` register.

4. After each write access to the `UxDBUF` register, poll `IRCON2.UTXxIF` to wait until the UART has transmitted the last data.

```c
// C language code:
// The two functions below send a range of bytes on the UARTx TX line. Note
// that, before the relevant function is called the application must execute
// the initialization code in Figure 3, Figure 11, Figure 12, and Figure 13.

// The code implements the following steps:
// 1. Clear TX interrupt request (UTXxIF = 0).
// 2. Loop: send each UARTx source byte on the UARTx TX line.
// 2a. Read byte from the allocated UART TX source buffer and write to UxDBUF.
// 2b. Wait until UART byte has been sent (UTXxIF = 1).
// 2c. Clear UTXxIF.

void uart0Send(unsigned short* uartTxBuf, unsigned short uartTxBufLength) {
  unsigned short uartTxIndex;
  UTX0IF = 0;
  for (uartTxIndex = 0; uartTxIndex < uartTxBufLength; uartTxIndex++) {
    U0DBUF = uartTxBuf[uartTxIndex];
    while( !UTX0IF );
    UTX0IF = 0;
  }
}


void uart1Send(unsigned short* uartTxBuf, unsigned short uartTxBufLength) {
  unsigned short uartTxIndex;
  UTX1IF = 0;
  for (uartTxIndex = 0; uartTxIndex < uartTxBufLength; uartTxIndex++) {
    U1DBUF = uartTxBuf[uartTxIndex];
    while( !UTX1IF );
    UTX1IF = 0;
  }
}
```

Figure 14: UART TX Processing using UART Polling

4.2.1.2 UART RX processing using UART Polling

The required code for UART RX, with polling, is shown in Figure 15, and implements the following main steps:

1. Allocate UART RX buffer for the associated target data.

2. Disable the UART RX interrupt (this instruction will not mask the generation of the interrupt request flags, used for polling UART RX complete; ref. Figure 2).

3. Enable UART RX.

4. Poll `URXxIF` to wait until the UART has received the byte.

5. Read the appropriate `UxDBUF` register, and store the contents in the allocated UART RX buffer.

```
// C language code:

// The two functions below receive a range of bytes on the UARTx RX line.
// Note that, before this function is called the application must execute
// the UART initialization code in Figure 3, Figure 11, Figure 12, and
// Figure 13.

// The code implements the following steps:
// 1. Enable UARTx RX (UxCSR.RE = 1)
// 2. Clear RX interrupt request (set URXxIF = 0)
// 3. Loop: receive each UARTx sample from the UARTx RX line.
// 3a. Wait until data received (URXxIF = 1).
// 3b. Read UxDBUF and store the value in the allocated UART RX target buffer.

void uart0Receive(unsigned short* uartRxBuf, unsigned short uartRxBufLength) {
  unsigned short uartRxIndex;

  U0CSR |= 0x40; URX0IF = 0;
  for (uartRxIndex = 0; uartRxIndex < uartRxBufLength; uartRxIndex++) {
    while( !URX0IF );
    uartRxBuf[uartRxIndex] = U0DBUF;
    URX0IF = 0;
  }

}


void uart1Receive(unsigned short* uartRxBuf, unsigned short uartRxBufLength) {
  unsigned short uartRxIndex;

  U1CSR |= 0x40; URX1IF = 0;
  for (uartRxIndex = 0; uartRxIndex < uartRxBufLength; uartRxIndex++) {
    while( !URX1IF );
    uartRxBuf[uartRxIndex] = U1DBUF;
    URX1IF = 0;
  }

}
```

Figure 15: UART RX Processing using UART Polling

4.2.2 Processing UART Communication using UART ISR

The following section shows how to control the UART TX/RX communication, involving a designated UART ISR. This method typically implies that the application just needs to initiate/start the UART TX/RX session, and then the remaining transmission/reception is processed automatically by the UART ISR. Consequently the application/CPU can continue both polling tasks, as well as interrupt tasks while the UART communication runs.

4.2.2.1 UART TX Processing using UART ISR

The required code for UART TX, with ISR, is shown in Figure 16, and implements the following main steps:

1. Allocate UART TX buffer + index for the associated source data.

2. Enable the UART TX interrupt. This instruction will allow the CPU to vector its execution to the designated UART TX ISR shown in Figure 17 (refer to Figure 2 for interrupt mapping).

3. Send the very first byte of the allocated UART TX buffer on UART.

4. Apply code in Figure 17 to send the remaining data using UART TX ISR.

```
// C language code:
// This function starts the UART TX session and leaves the transmission
// of the remaining bytes to the associated UART TX ISR in Figure 17.
// Before this function is called the application must initialize the
// UART peripheral according to the code shown in Figure 3, Figure 11,
// Figure 12, and Figure 13.

// The code implements the following steps:
// 1. Initialize the UART TX buffer index.
// 2. Clear UART TX Interrupt Flag (IRCON2.UTXxIF = 0.
// 3. Enable UART TX Interrupt (IEN2.UTXxIE = 1)
// 4. Send very first UART byte
// 5. Enable global interrupt (IEN0.EA = 1).

extern unsigned short __xdata uartTxBuffer[SIZE_OF_UART_TX_BUFFER];
extern unsigned short __xdata uartTxIndex;

void uartStartTxForIsr(unsigned char uartNum) {

  uartTxIndex = 0;

  if (uartNum == 0) {
    UTX0IF = 0;
    IEN2 |= 0x04;
    U0DBUF = uartTxBuffer[uartTxIndex++];
  } else {
    UTX1IF = 0;
    IEN2 |= 0x08;
    U1DBUF = uartTxBuffer[uartTxIndex++];
  }

  IEN0 |= 0x80;
}
```

Figure 16: Initializing UART TX Processing for UART ISR

```
// C language code:

// The UARTx TX ISR assumes that the code in Figure 16 has initialized the
// UART TX session, by sending the very first byte on the UARTx TX line.
// Then the UARTx TX ISR will send the remaining bytes based in interrupt
// request generation by the UART peripheral.

// The code implements the following steps:
// 1. Clear UARTx TX Interrupt Flag (IRCON2.UTXxIF = 0).
// 2. Read byte from the allocated UART TX source buffer and write to UxDBUF.
// 3. If no UART byte left to transmit, stop this UART TX session.
//    Note that in order to start another UART TX session the application
//    just needs to prepare the source buffer, and simply send the very first
//    UARTx byte.

extern unsigned short __xdata uartTxBuffer[SIZE_OF_UART_TX_BUFFER];
extern unsigned short __xdata uartTxIndex;

_Pragma("vector=0x3B") __near_func __interrupt void UART0_TX_ISR(void);
_Pragma("vector=0x3B") __near_func __interrupt void UART0_TX_ISR(void){

  UTX0IF = 0;

  if (uartTxIndex >= SIZE_OF_UART_TX_BUFFER) {
    uartTxIndex = 0; IEN2 &= ~0x08; return;
  }

  U0DBUF = uartTxBuffer[uartTxIndex++];
}


_Pragma("vector=0x73") __near_func __interrupt void UART1_TX_ISR(void);
_Pragma("vector=0x73") __near_func __interrupt void UART1_TX_ISR(void){

  UTX1IF = 0;

  if (uartTxIndex >= SIZE_OF_UART_TX_BUFFER) {
    uartTxIndex = 0; IEN2 &= ~0x08; return;
  }

  U1DBUF = uartTxBuffer[uartTxIndex++];
}
```

Figure 17: UART TX Processing in the UART ISR

4.2.2.2 UART RX Processing using UART ISR

The required code for UART RX, with ISR, is shown in Figure 18 and implements the following main steps:

1. Allocate UART RX buffer + index for the associated target data.

2. Enable UART RX.

3. Enable the UART RX interrupt to allow the CPU to automatically vector its execution to the designated UART RX ISR, shown in Figure 19, once the UART has received a byte (refer to Figure 2 for interrupt mapping).

4. Receive the bytes using UART RX ISR, as shown in Figure 19.

```
// C language code:
// This function initializes the UART RX session, by simply enabling the
// corresponding UART interrupt, and leave the sample reception to the
// UART ISR shown in Figure 19. Before this function is called the
// application must initialize the UART peripheral according to the
// code shown in Figure 3, Figure 11, Figure 12, and Figure 13.

// The code implements the following steps:
// 1. Initialize the UART RX buffer index.
// 2. Clear UART RX Interrupt Flag (TCON.URXxIF = 0)
// 3. Enable UART RX and Interrupt (IEN0.URXxIE = 1, UxCSR.RE = 1)
// 4. Enable global interrupt (IEN0.EA = 1)

extern unsigned short __xdata uartRxBuffer[SIZE_OF_UART_RX_BUFFER];
extern unsigned short __xdata uartRxIndex;

void uartStartRxForIsr(unsigned char uartNum) {

  uartRxIndex = 0;

  if (uartNum == 0) {
    URX0IF = 0;
    U0CSR |= 0x40;
    IEN0 |= 0x04;
  } else {
    URX1IF = 0;
    U1CSR |= 0x40;
    IEN0 |= 0x08;
  }

  IEN0 |= 0x80;
}
```

Figure 18: Initializing UART RX Processing for the UART ISR

```
// C language code:

// The UARTx RX ISR assumes that the code in Figure 18 has initialized the
// UART RX session, by enabling the UART RX interrupt. Then this UART RX ISR
// will receive the data based in interrupt request generation by the
// USART peripheral.

// The code implements the following steps:
// 1. Clear UARTx RX Interrupt Flag (TCON.URXxIF = 0)
// 2. Read UxDBUF and store the value in the allocated UART RX target buffer
// 3. If all UART data received, stop this UART RX session
//    Note that in order to start another UART RX session the application
//    just needs to re-enable the UART RX interrupt(IEN0.URXxIE = 1).

extern unsigned short __xdata uartRxBuffer[SIZE_OF_UART_RX_BUFFER];
extern unsigned short __xdata uartRxIndex;

_Pragma("vector=0x13") __near_func __interrupt void UART0_RX_ISR(void);
_Pragma("vector=0x13") __near_func __interrupt void UART0_RX_ISR(void){

  URX0IF = 0;

  uartRxBuffer[uartRxIndex++] = U0DBUF;

  if (uartRxIndex >= SIZE_OF_UART_RX_BUFFER) {
    uartRxIndex = 0; IEN0 &= ~0x04;
  }

}


_Pragma("vector=0x1B") __near_func __interrupt void UART1_RX_ISR(void);
_Pragma("vector=0x1B") __near_func __interrupt void UART1_RX_ISR(void){

  URX1IF = 0;

  uartRxBuffer[uartRxIndex++] = U1DBUF;

  if (uartRxIndex >= SIZE_OF_UART_RX_BUFFER) {
    uartRxIndex = 0; IEN0 &= ~0x08;
  }

}
```

Figure 19: UART RX Processing in the UART ISR

4.3 Using the UART with DMA Support

In order to use the DMA controller to support the UART, typically two DMA channels must be allocated and configured; one for downloading data from SoC memory to the UART (TX), and another for uploading data from the UART to SoC memory (RX). Please refer to the “DMA Controller” section in the SoC Data Sheets ([1] and [2]) for more detailed information about the DMA controller.

4.3.1 Allocating DMA Descriptor for UART RX/TX

Before the UART can transmit/receive with DMA support, the application must allocate associated DMA descriptors, that is; one for RX, and one for TX. The required code is shown in Figure 20:

```
// C language code:

// Define data structure for DMA descriptor:
typedef struct {
  unsigned char SRCADDRH;       // High byte of the source address
  unsigned char SRCADDRL;       // Low byte of the source address
  unsigned char DESTADDRH;      // High byte of the destination address
  unsigned char DESTADDRL;      // Low byte of the destination address
  unsigned char VLEN      : 3;  // Length configuration
  unsigned char LENH      : 5;  // High byte of fixed length
  unsigned char LENL      : 8;  // Low byte of fixed length
  unsigned char WORDSIZE  : 1;  // Number of bytes per transfer element
  unsigned char TMODE     : 2;  // DMA trigger mode (e.g. single or repeated)
  unsigned char TRIG      : 5;  // DMA trigger; UART RX/TX
  unsigned char SRCINC    : 2;  // Number of source address increments
  unsigned char DESTINC   : 2;  // Number of destination address increments
  unsigned char IRQMASK   : 1;  // DMA interrupt mask
  unsigned char M8        : 1;  // Number of desired bit transfers in byte mode
  unsigned char PRIORITY  : 2;  // The DMA memory access priority
} DMA_DESC;


// Allocate DMA descriptor for UART RX/TX:
// Note that, since the DMA controller only offers one address/reference
// register for DMA channels 1 - 4, the DMA controller expects the
// allocated descriptors for DMA channels 2 – 4 to be located in direct
// address succession to the DMA channel 1 descriptor. This is typically
// relevant when the application has already allocated DMA channel 0, and 1,
// for other purposes than UART support.
DMA_DESC __xdata uartDmaRxTxCh[2];
```

Figure 20: Allocating Buffers and DMA Descriptors for UART RX/TX

4.3.2 Processing UART Communication with DMA Support

The following section shows how to control the UART TX/RX communication using DMA support. This method typically implies that the application just needs to initiate/start the UART TX/RX session, and then the remaining byte transfers are handled automatically by the DMA controller. The application/CPU can continue both polling tasks, as well as interrupt tasks while the UART communication goes on.

The required code for UART TX/RX, using DMA support (shown in Figure 20, Figure 21, Figure 22, and Figure 23) implements the following main steps:

1. Allocate UART TX/RX buffer + index for the associated source/target data.

2. Define and allocate data structures for DMA channel configuration (descriptors); used to setup a DMA channel for transferring data between SoC memory and the UART.

3. Start the UART TX session by applying a manual DMA trigger to the associated DMA channel. This triggers the DMA controller to download a byte from SoC memory to the `UxDBUF` register. Once the UART has transmitted the byte it automatically triggers another DMA byte transfer. This sequence continues until the DMA controller has completed the programmed number of byte transfers. In order to start the UART RX session the application only needs to arm the DMA channel associated with UART RX. The byte transfers from the UART to SoC memory are triggered by the UART; every time it has either received a UART byte.

4. Once the DMA controller has completed the defined range of byte transfers, the application can start another UART RX session, by simply re-arming the associated DMA channel. For UART TX the DMA controller needs a manual trigger (alternatively, the application/CPU could write the very first UART TX data to the `UxDBUF` register. For efficient data streaming (typically between RF and UART) the application could implement the UART re-start mechanism in e.g. a designated DMA ISR, as shown in Figure 23, thus isolating the UART TX/RX session as much as possible from the main application code.

```
// C language code:
// This function sets up a designated DMA channel for UART TX and starts
// the UART TX session. Before this function is called the application must
// perform initialization by executing the code in Figure 3, Figure 11,
// Figure 12, and Figure 13.

void uartStartTxDmaChan( unsigned char uartNum,
                         DMA_DESC *uartDmaTxDescr,
                         unsigned char uartDmaTxChan,
                         unsigned char* uartTxBuf,
                         unsigned short uartTxBufSize) {

  // Source = allocated UART TX buffer, destination = UxDBUF
  // Number of DMA byte transfers = UART TX buffer size.
  uartDmaTxDescr->SRCADDRH   = (unsigned short)(uartTxBuf+1)>>8;
  uartDmaTxDescr->SRCADDRL   = (unsigned short)(uartTxBuf+1);
  uartDmaTxDescr->DESTADDRH  = 0xDF;
  uartDmaTxDescr->DESTADDRL  = (uartNum == 0) ? 0xC1:0xF9;
  uartDmaTxDescr->LENH       = ((uartTxBufSize-1)>>8)&0xFF;
  uartDmaTxDescr->LENL       = (uartTxBufSize-1)&0xFF;

  uartDmaTxDescr->VLEN       = 0x00;  // Use fixed length DMA transfer count
  uartDmaTxDescr->WORDSIZE   = 0x00;  // Perfrom 1-byte DMA transfers
  uartDmaTxDescr->TMODE      = 0x00;  // Single byte transfer per DMA trigger
  uartDmaTxDescr->TRIG = 15 + (2*uartNum);  // DMA trigger = USARTx TX complete
  uartDmaTxDescr->SRCINC     = 0x01;  // Increment source pointer by 1 byte
                                      // address after each transfer.
  uartDmaTxDescr->DESTINC    = 0x00;  // Do not increment destination pointer:
                                      // points to USART UxDBUF register.
  uartDmaTxDescr->IRQMASK    = 0x01;  // Enable DMA interrupt to the CPU
  uartDmaTxDescr->M8         = 0x00;  // Use all 8 bits for transfer count
  uartDmaTxDescr->PRIORITY   = 0x00;  // DMA memory access has low priority

  // Link DMA descriptor with its corresponding DMA configuration register.
  if (uartDmaTxChan < 1) {
    DMA0CFGH = (unsigned char)((unsigned short)uartDmaTxDescr>>8);
    DMA0CFGL = (unsigned char)((unsigned short)uartDmaTxDescr&0x00FF);
  } else {
    DMA1CFGH = (unsigned char)((unsigned short)uartDmaTxDescr>>8);
    DMA1CFGL = (unsigned char)((unsigned short)uartDmaTxDescr&0x00FF);
  }

  // Arm DMA channel and apply 45 NOP's for loading DMA configuration
  DMAARM = ((1 << uartDmaTxChan) & 0x1F);
  asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
  asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
  asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
  asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
  asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
  asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
  asm("NOP");asm("NOP");asm("NOP");

  // Enable the DMA interrupt (IEN1.DMAIE = IEN0.EA = 1),
  // and clear potential pending DMA interrupt requests (IRCON.DMAIF = 0).
  IEN0 |= 0x80; IEN1 |= 0x01; IRCON &= ~0x01;

  // Send the very first UART byte to trigger a UART TX session:
  if (uartNum == 0) {
    U0DBUF = uartTxBuf[0];
  } else {
    U1DBUF = uartTxBuf[0];
  }

  // At this point the UART generates a DMA trigger each time it has
  // transmitted a byte, leading to a DMA transfer from the allocated
  // TX source buffer to UxDBUF. Once the DMA controller has completed
  // the defined range of transfers, the CPU vectors its execution to
  // the DMA ISR in Figure 23.
}
```

Figure 21: UART TX Initialization using DMA Support

```
// C language code:
// This function a designated DMA channel for UART RX and then enables
// the UART RX session. Before this function is called the application must
// perform initialization by executing the code in Figure 3, Figure 11,
// Figure 12, and Figure 13.

void uartStartRxDmaChan( unsigned char uartNum,
                         DMA_DESC *uartDmaRxDescr,
                         unsigned char uartDmaRxChan,
                         unsigned char* uartRxBuf,
                         unsigned short uartRxBufSize) {

  // Source = UxDBUF, destination = allocated UART RX buffer
  // Number of DMA byte transfers = UART RX buffer size.
  uartDmaRxDescr->DESTADDRH  = (unsigned short)uartRxBuf>>8;
  uartDmaRxDescr->DESTADDRL  = (unsigned short)uartRxBuf;
  uartDmaRxDescr->SRCADDRH   = 0xDF;
  uartDmaRxDescr->SRCADDRL   = (uartNum == 0) ? 0xC1:0xF9;
  uartDmaRxDescr->LENH       = (uartRxBufSize>>8)&0xFF;
  uartDmaRxDescr->LENL       = uartRxBufSize&0xFF;

  uartDmaRxDescr->VLEN       = 0x00;  // Use fixed length DMA transfer count
  uartDmaRxDescr->WORDSIZE   = 0x00;  // Perform 1-byte transfers
  uartDmaRxDescr->TMODE      = 0x00;  // Single byte transfer per DMA trigger
  uartDmaRxDescr->TRIG = 14 + (2*uartNum);  // DMA trigger = USARTx RX complete
  uartDmaRxDescr->SRCINC     = 0x00;  // Do not increment source pointer.
                                      // points to USART UxDBUF register.
  uartDmaRxDescr->DESTINC    = 0x01;  // Increment destination pointer by
                                      // 1 byte address after each transfer.
  uartDmaRxDescr->IRQMASK    = 0x01;  // Enable DMA interrupt to the CPU
  uartDmaRxDescr->M8         = 0x00;  // Use all 8 bits for transfer count
  uartDmaRxDescr->PRIORITY   = 0x00;  // DMA memory access has low priority

  // Link DMA descriptor with its corresponding DMA configuration register.
  if (uartDmaRxChan < 1) {
    DMA0CFGH = (unsigned char)((unsigned short)uartDmaRxDescr>>8);
    DMA0CFGL = (unsigned char)((unsigned short)uartDmaRxDescr&0x00FF);
  } else {
    DMA1CFGH = (unsigned char)((unsigned short)uartDmaRxDescr>>8);
    DMA1CFGL = (unsigned char)((unsigned short)uartDmaRxDescr&0x00FF);
  }

  // Arm DMA channel and apply 45 NOP's for loading DMA configuration
  DMAARM = ((1 << uartDmaRxChan) & 0x1F);
  asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
  asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
  asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
  asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
  asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
  asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
  asm("NOP");asm("NOP");asm("NOP");

  // Enable the DMA interrupt (IEN1.DMAIE = IEN0.EA = 1),
  // and clear potential pending DMA interrupt requests (IRCON.DMAIF = 0)
  IEN0 |= 0x80; IEN1 |= 0x01; IRCON &= ~0x01;

  // Enable UARTx RX
  if (uartNum == 0) {
    U0CSR |= 0x40;
  } else {
    U1CSR |= 0x40;
  }

  // At this point the UART generates a DMA trigger each time it has received
  // a byte, leading to a DMA transfer from UxDBUF to the allocated RX target
  // buffer. Once the DMA controller has completed the defined range of
  // transfers, the CPU vectors its execution to the DMA ISR in Figure 24.
}
```

Figure 22: UART RX Initialization using DMA Support

```
// C language code:
// This DMA ISR can be used to start a new UART TX session when the previous
// session (started by the code in Figure 21 or Figure 22) has completed.
// For simplicity the code assumes that DMA channel 1 is used, but the
// functionality is the same for other DMA channels.

// The code implements the following steps:
// 1. Clear the main DMA interrupt Request Flag (IRCON.DMAIF = 0).
// 2. Start a new UART TX session on the applied DMA channel.
// 2a. Clear DMA Channel Interrupt Request Flag (DMAIRQ.DMAIFx = 0).
// 2b. Re-arm the applied DMA Channel (DMAARM.DMAARMx = 1).
// 2c. Send the very first UART TX byte to trigger a new UART TX session.

external UART_PROT_CONFIG __xdata uartProtConfig;

_Pragma("vector=0x43") __near_func __interrupt void DMA_ISR(void);
_Pragma("vector=0x43") __near_func __interrupt void DMA_ISR(void) {
  IRCON &= ~0x01;

  if (DMAIRQ & 0x02) {
    DMAIRQ &= ~0x02;

    // Here the application could typically perform a quick preparation of
    // the allocated UART TX source buffer before starting another UART TX
    // session.

    // Recommendation:
    // Introduce delay here to allow receiver processing between DMA packets.
    // The applied delay should then be tuned according to UART data rate.

    DMAARM |= 0x02;

    if (uartProtConfig.uartNum == 0) {
      U0DBUF = uartTxBuffer[0];
    } else {
      U1DBUF = uartTxBuffer[0];
    }
  }

}
```

Figure 23: UART TX re-start using DMA ISR

```
// C language code:
// This DMA ISR can be used to start a new UART RX session when the previous
// session (started by the code in Figure 21 or Figure 22) has completed.
// For simplicity the code assumes that DMA channel 0 is used, but the
// functionality is the same for other DMA channels.

// The code implements the following steps:
// 1. Clear the main DMA interrupt Request Flag (IRCON.DMAIF = 0).
// 2. Start a new UART RX session on the applied DMA channel.
// 2a. Clear applied DMA Channel Interrupt Request Flag (DMAIRQ.DMAIFx = 0).
// 2b. Re-arm applied DMA Channel (DMAARM.DMAARMx = 1).

external UART_PROT_CONFIG __xdata uartProtConfig;

_Pragma("vector=0x43") __near_func __interrupt void DMA_ISR(void);
_Pragma("vector=0x43") __near_func __interrupt void DMA_ISR(void) {
  IRCON &= ~0x01;

  if (DMAIRQ & 0x01) {

    // Here the application could typically perform a quick initial check
    // of the received data before re-starting another UART RX session.

    DMAIRQ &= ~0x01;
    DMAARM |= 0x01;

  }

}
```

Figure 24: UART RX re-start using DMA ISR

5 References

[1] CC1110Fx/CC1111Fx Data Sheet (SWRS033)

[2] CC2510Fx/CC2511Fx Data Sheet (SWRS055)

[3] CC2430 Data Sheet (SWRS036)

[4] CC2530 Data Sheet(SWRS081)

[5] UART Communication (Wikipedia - UART)

[6] RS232 Signaling (Wikipedia - RS232 Signaling)

[7] RS232 Serial Cables (Wikipedia - RS232 Serial Cables)

6 General Information

6.1 Document History

Revision	Date	Description/Changes
SWRA222B	2009.06.30	Updated for CC2530 and CC2531.
SWRA222A	2008.09.26	Corrected name+contents of UART protocol initialization structure; replaced UART_PROT_DESC with UART_PROT_CONFIG.
SWRA222	2008.08.13	Initial release.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265