Application Note

AN2371/D
Rev. 0, 10/2002

Using the Universal Asynchronous Receiver Transmitter TPU Function (UART) with the MPC500 Family

Vernon Goler
TECD

This TPU Programming Note is intended to provide simple C interface routines to the asynchronous receiver transmitter TPU function (UART). [1] The routines are targeted for the MPC500 family of devices but, they should be easy to use with any device that has a TPU.

1 Functional Overview

The UART function uses two TPU channels to provide a 3-wire (TxD, RxD and GND) asynchronous serial interface. One TPU channel is configured to function as the serial transmitter (TxD), and another TPU channel is configured to function as a serial receiver (RxD). All standard baud rates and parity checking can be selected.

2 Detailed Description

A UART consists of a transmitter, which can transmit serial data via a transmit data (TxD) pin, and a receiver, which can receive serial data via a receive data (RxD) pin. Both transmitter and receiver contain a single shift register that performs parallel-to-serial and serial-to-parallel conversion. Although a UART IC normally contains both the transmitter and receiver, implementing a full-duplex UART with the TPU requires independent receiver and transmitter channels because each TPU channel controls only one pin (a channel can be either a transmitter or a receiver, but not both at the same time). While at least two TPU channels must be used for a fully functional UART, it is not necessary to use both subfunctions together, nor to use the same number of receivers and transmitters. For example, the TPU can be configured to function as 13 transmitters and 3 receivers. There is also no restriction on which channels may be used to receive and transmit - any channel can be used to transmit or receive data. Since baud rate for each channel is specified independently, a transmitter can have a different rate than a receiver

The UART protocol allows selection of a parity bit to detect simple transmission errors. Parity can be generated and checked in three different ways: odd, even and no parity. All parity types are supported with the UART function.

The UART protocol is not fixed to a specific number of bits for one data word. Although 8-bit words are normally used, some applications use 7-bit or 9-bit words. The UART function can

[1]The information in this Programming Note is based on TPUPN07. It is intended to compliment the information found in that Programming Note.

use word lengths from one to 14 bits. The number of transmitter stop bits is fixed at one. The receiver can also handle fractional stop bits correctly, but the transmitter cannot generate fractional stop bits.

The UART function is double buffered. Both transmitter and receiver contain a shift register as well as a data register. The host CPU can write new data to the transmit data register while data is being transmitted, and can read data from the receive data register while data is being received.

The UART transmitter sets the channel interrupt status flag to indicate when all the data has been transmitted. The status flag should be cleared before new data is written to the transmitter if the interrupt status flag will be used in a polling environment. Likewise the UART receiver sets the channel interrupt status flag to indicate the arrival of new data. The status flag should be cleared after reading the received data if the interrupt status flag will be used in a polling environment. The detection of new received data, reading the data, and clearing the status flag must complete before new data is received to avoid missing data, or possibly reading the same data twice.

The UART function can do back-to-back transfers. If data is available in time, the transmitter does not generate an idle line signal, but transmits exactly one stop bit followed by the start bit for the next data. An idle line condition only occurs if the transmit data register is empty after transferring data. The length of a transmit idle line condition is always divisible by the baud_rate parameter. The receiver can handle any length of idle line.

Every data word begins with one start bit, which is always a logic zero. Following the start bit, a specified number of data bits are transmitted least significant bit first, then a parity bit is generated and transmitted if parity is enabled. The end of the data word is marked by one stop bit, which is always a logic one. An idle line consists of successive stop bits, which means that the line is at logic level one while idle. For example the ASCII character “A” is always transmitted as %0(start bit) 0100 0001(“A”) (parity bit if enabled) 1(stop bit).

This note uses the term “bit time” to refer to the time required to transmit or receive one bit. Bit time is determined by baud rate, using the formula:

Bit Time = 1/Baud Rate

The receiver detects a data word by sensing the falling edge of the start bit. Since the UART function always treats the first falling edge after the initialization service request as a valid start bit, a receiver must be enabled only when the line is idle. A received bit is sampled only once, approximately halfway through the bit time.

2.1 UART C Level API

Rather then controlling the TPU registers directly, the UART routines in this TPU Programming Note may be used to provide a simple and easy interface. There are 4 routines for controlling the UART function in 2 files (tpu_uart.h and tpu_uart.c). The tpu_uart.h file should be included in any files that use the routines. This files contains the function prototypes and useful #defines. Each of the routines in tpu_uart.c will be described in detail. The routines are:

- Initialization Functions:
  - void tpu_uart_transmit_init(struct TPU3_tag *tpu, UINT8 channel, UINT8 priority, INT16 baud_rate, INT16 bits_per_data_word, UINT8 parity, UINT8 nointerrupt_interrupt);
  - void tpu_uart_receive_init(struct TPU3_tag *tpu, UINT8 channel, UINT8 priority, INT16 baud_rate, INT16 bits_per_data_word, UINT8 parity, UINT8 nointerrupt_interrupt);
- Functions to write data to transmit and read received data:

— void tpu_uart_write_transmit_data(struct TPU3_tag *tpu, UINT8 channel, INT16 transmit_data);

— void tpu_uart_read_receive_data(struct TPU3_tag *tpu, UINT8 channel, INT16 *receive_data, UINT8 *parity_error, UINT8 *framing_error);

2.1.1 void tpu_uart_transmit_init

This function is used to initialize a channel to run the transmit UART function. This function has 7 parameters:

- *tpu - This is a pointer to the TPU3 module to use. It is of type TPU3_tag which is defined in m_tpu3.h.
- channel - This is the TPU channel number of the UART channel. This parameter should be assigned a value of 0 to 15.
- priority - This is the priority to assign to the channel. This parameter should be assigned a value of TPU_PRIORITY_HIGH, TPU_PRIORITY_MIDDLE or TPU_PRIORITY_LOW. The TPU priorities are defined in mpc500_utils.h.
- baud_rate – Baud rate is a measure of the number of times per second a signal in a communications channel varies, or makes a transition between states (states being frequencies, voltage levels, or phase angles). One baud is one such change. Thus, a 300-baud modem's signal changes state 300 times each second, while a 600- baud modem's signal changes state 600 times per second. The baud_rate value is the number of TCR1 counts per bit time, and is calculated by the following equation:

$$\frac{\text{Timer Count Register (TCR1) counts/second}}{\text{number of transitions baud/second}}$$

- bits_per_data_word – This is the number of bits to be transmitted in one data word. This bits_per_data_word commonly has a value of eight, because most serial protocols use 8-bit words.
- parity – This is the desired parity. This parameter should be assigned a value of TPU_UART_NOPARITY, TPU_UART__EVEN_PARITY, or TPU_UART__ODD_PARITY. The TPU parity for the UART function is defined in tpu_uart.h.
- nointerrupt_interrupt – This parameter determines whether an interrupt is generated after each data word is transmitted.

2.1.2 void tpu_uart_receive_init

This function is used to initialize a channel to run the receive UART function. This function has 7 parameters:

- *tpu - This is a pointer to the TPU3 module to use. It is of type TPU3_tag which is defined in m_tpu3.h.
- channel - This is the TPU channel number of the UART channel. This parameter should be assigned a value of 0 to 15.
- priority - This is the priority to assign to the channel. This parameter should be assigned a value of TPU_PRIORITY_HIGH, TPU_PRIORITY_MIDDLE or TPU_PRIORITY_LOW. The TPU priorities are defined in mpc500_utils.h.

- baud_rate – Baud rate is a measure of the number of times per second a signal in a communications channel varies, or makes a transition between states (states being frequencies, voltage levels, or phase angles). One baud is one such change. Thus, a 300-baud modem's signal changes state 300 times each second, while a 600- baud modem's signal changes state 600 times per second. The baud_rate value is the number of TCR1 counts per bit time, and is calculated by the following equation:

$$\frac{\text{Timer Count Register (TCR1) counts/second}}{\text{number of transitions baud/second}}$$

- bits_per_data_word – This is the number of bits to be received in one data word. This bits_per_data_word commonly has a value of eight, because most serial protocols use 8-bit words.
- parity – This is the desired parity. This parameter should be assigned a value of TPU_UART_NOPARITY, TPU_UART__EVEN_PARITY, or TPU_UART__ODD_PARITY. The TPU parity for the UART function is defined in tpu_uart.h.
- nointerrupt_interrupt – This parameter determines whether an interrupt is generated when each data word is received.

NOTE

Care should be taken when initializing TPU channels. The TPU's behavior may become unpredictable if a channel is reinitialized while it is running. This unpredictability can occur because there is no way to stop a TPU channel that is executing code. Therefore, the channel must complete the execution of code before it is reinitialized. To ensure that the channel is stopped before it is configured, the channel's priority should be set to disabled. If the channel is currently being serviced when the priority is set to disabled, it will continue to service the channel until the state ends. To ensure that the channel is not being serviced, the user should wait for the longest state execution time after disabling the channel. All channels are disabled out of reset so that they can be configured immediately from reset.

The *tpu_uart_init* function attempts to wait between the disabling of the channels before it starts configuring them, however the actual execution speed of the code will be depend on the specific system. If you are not configuring the channels from reset, then ideally it is best to have the functions disabled before calling this function. TPU channels can be disabled by using the *tpu_disable* function in the mpc500_utils.c file. For example, disabling channel 0 is done like this: tpu_disable(tpu, 0);

2.1.3 void tpu_uart_write_transmit_data

This function is used to send the data that is to be serially transmitted. This function has 3 parameters:

- *tpu – This is a pointer to the TPU3 module to use. It is of type TPU3_tag which is defined in m_tpu3.h
- channel – This is the TPU channel number of the UART channel. This parameter should be assigned a value of 0 to 15.
- transmit_data – This the actual data word to be transmitted. Up to 14 bits of data per data word can be transmitted.

2.1.4 void tpu_uart_read_receive_data

This function is used to get the serially received data. In addition, parity can be checked if enabled. Any framing errors can also be checked. Both parity and framing errors are only valid for each received data word. Each new word received will update both parity and framing error information, overwriting previous values. This function has 5 parameters:

- *tpu – This is a pointer to the TPU3 module to use. It is of type TPU3_tag which is defined in m_tpu3.h
- channel – This is the TPU channel number of the UART channel. This parameter should be assigned a value of 0 to 15.
- *receive_data – This is a pointer to the received data. The calling routine of this function should pass the address of where the received data is to be stored in the receive_data parameter. Up to 14 bits of data per data word can be received.
- *parity_error – If parity is enabled, this parameter is set to either one or zero depending on whether even or odd parity is enabled. For even parity this bit is set to a one if the number of ones in the received data word is odd, else the bit is set to zero. For odd parity this bit is set to a one if the number of ones in the received data word is even, else the bit is set to zero. The calling routine of this function should pass the address of where the parity error parameter is to be stored in *parity_error. The parity is only valid for each received data word.
- *framing_error – The framing_error parameter is set to a one if a framing error is detected. A framing error occurs when the UART function determines that a stop bit is low instead of high. The calling routine of this function should pass the address of where the framing error parameter is to be stored in *framing_error. The framing error indication is only valid for each received data word.

3 Asynchronous Serial Interface Example

The following example shows configuration of a TPU channel as a transmitter, and another TPU channel as a receiver. Test data is serial transmitted out of the TPU transmitter channel, and if the two channels are connected together serial data is received into the TPU receiver channel. This example is a C program that shows how to configure and use the UART interface routines.

3.1 Example 1

3.1.1 Description

This sample program show a simple UART example in which channel 0 is configured as a transmitter and channel 1 is configured as a receiver. When the two channels are physically connected together, each data word transmitted by channel 0 will be received by channel 1. The received data is then stored in an array to allow checking of the received data. The baud rate is set to approximately 9600 baud. The actual baud rate depends on the clock frequency of the MCU, and the prescaler value chosen.

3.1.2 Program

```
/*************************************************************************************/
/* FILE NAME: tpu_uart_ex1.c  COPYRIGHT (c)  2002                                    */
/* VERSION: 1.0               All Rights Reserved                                    */
/*                                                                                   */
/* DESCRIPTION: This routine is used to initialize TPU channel 0 to                  */
/* run the UART transmit function and  to send out a test stream of data.            */
/* Channel 1 is initialized to run the UART receive function and to receive the      */
/* test stream of data if channel 0 and 1 are physically connected together.         */
/* The received data is stored in array  store_receive_data_pointer []               */
/*                                                                                   */
/*                                                                                   */
/*=====================================================================              */
/* HISTORY                    ORIGINAL AUTHOR: Vernon Goler                          */
/* REV              AUTHOR              DATE              DESCRIPTION OF CHANGE      */
/* ---              ------              ----              ---------------------      */
/* 1.0              V. GOLER            15/SEP/02         Initial Version of Function */
/*************************************************************************************/


#include "mpc555.h"
#include "mpc500_util.h"
#include "tpu_uart.h"

#define UART_BAUD_RATE       0x01b2              /* set the baud rate, ~9600 baud       */
#define UART_DATA_SIZE       0x0008              /* set data size of transmit and receive data*/



void main()
{

struct TPU3_tag *tpua = &TPU_A;                            /* pointer for TPU routines   */

UINT8 parity_error;                                        /* parity error flag          */
UINT8 framing_error;                                       /* framing error flag         */
INT16 receive_data;                                        /* received data              */

char test_message[] = "1 2 3 4 5 6 7 8 9 This is a test of the TPU transmitter function";
char store_receive_data_pointer[100];            /* place to store received data        */

int i;                                           /* index variable                      */
```

```
          /* initialize channel 0 to act as a transmitter                                 */
          tpu_uart_transmit_init(tpua, 0, TPU_PRIORITY_HIGH, UART_BAUD_RATE, UART_DATA_SIZE, \
          TPU_UART_NOPARITY, TPU_UART_NOINTERRUPT);

          /* initialize channel 1 to act as a receiver                                    */
          tpu_uart_receive_init(tpua, 1, TPU_PRIORITY_HIGH, UART_BAUD_RATE, UART_DATA_SIZE, \
          TPU_UART_NOPARITY, TPU_UART_NOINTERRUPT);



          for(i=0; test_message[i] != EOF; i++) {
              tpu_uart_write_transmit_data(tpua, 0, test_message[i]);          /* send data to be
transmitted        */

              while((tpu_check_interrupt(tpua, 0)) != 1)    /* wait for data to be transmitted*/
                    {
                    }

              while((tpu_check_interrupt(tpua, 1)) != 1)/* check for received data available */
                    {
                    }

          /* read received data, check for parity and framing error                      */
              tpu_uart_read_receive_data(tpua, 1, &receive_data, &parity_error, &framing_error);

              tpu_clear_interrupt(tpua, 1);       /* clear receive interrupt status channel 1*/

          /* save received data read in memory, mask to eight bit data          */
              store_receive_data_pointer[i] = receive_data;
          }

          while(1)
          {
          }                                                 /* wait after message transmitted*/

}
```

4 Function State Timing

When calculating the worst case latency for the TPU, the execution time of each state of the TPU is needed. The state timings for each of the six modes of the UART function are shown below in Table 1. The states used by the C interface functions are shown in Table 2.

Table 1. UART Function State Timing

State Number and Name	Max. CPU Clock Cycles	RAM Accesses by TPU
S0 INIT_RECEIVER	4	0
S1 INIT_TRANSMITTER	4	1
S2 POLLING_TDRE		7
(Transmitter only)		
TDRE = 1	16	
TDRE = 0	22	
S3 SENDING_DATA (transmitter only)		8
Transmit stop bit with parity	12	
Transmit stop bit no parity	20	
Transmit parity	32	
Transmit one data bit	28	
S4 RECEIVING_START_BIT (receiver only)		2
No parity selected	16	
Parity selected	18	
S5 RECEIVING_START_BIT (receiver only)		8
Receive stop bit	44 + (2 * (16 – data_size))	
Receive one data bit	20	

NOTE: Execution times do not include the time slot transition time (TST= 10 or 14 CPU clocks)

Table 2. UART API Function State Usage

UART API Function	State Uses
tpu_uart_transmit_init	S0
tpu_uart_receive_init	S1
tpu_uart_write_transmit_data	S2, S3
tpu_uart_read_receive_data	S4, S5

5 Function Code Size

Total TPU function code size determines what combination of functions can fit into a given ROM or emulation DPTRAM memory microcode space. UART function code size is:

59 µ instructions + 8 entries = 67 long words

6 Notes on Use and Performance of the UART Function

6.1 Performance

Like all TPU functions, the performance limit of the UART function in a given application is dependent to some extent on the service time (latency) associated with activity on other TPU channels. This is due to the operational nature of the scheduler. In the case of the UART function, this limits the maximum frequency and minimum pulse widths of the signal that can be properly measured.

Since the scheduler assures that the worst case latencies in any TPU application can be calculated, it is recommended that the guidelines given in the TPU reference manual are used along with the information given in the UART function state timing table to perform an analysis on any proposed TPU application that appears to approach the performance limits of the TPU.

To calculate the maximum performance of the function, the user must know the execution time for the different states of the functions running at the same time. In general, the maximum service latency for every mode of the UART function must be less than one bit time, which depends on the baud rate. The function must be allowed this amount of time by the other running functions.

6.1.1 Latency Example

The example assume that only the UART function is executing on a MPC555 running with a 40MHz system clock. The example is for an absolute worst case, e.g. all transmitters are transmitting parity bits for all bits transmitted.

When only transmitters are running, maximum baud rate for all channels combined is:

(time to switch channels – 10 clocks) + (time to transmit parity bit – 32 clocks) = 42 clocks/ bit

Forty-two clocks/bit rounds to 11 TCR1 counts, assuming that the resolution of each TCR1 count is 4 system clocks. There are 10,000,000 TCR1 counts/second for a 40 MHz system clock. 10,000,000 TCR1 counts/ second divided by 11 TCR1 counts/ bit = 909K bits/ second which is approximately 909 Kbaud for one transmitter. This translates to approximately 90 Kbaud for 10 transmitters.

Similar calculations can be performed for a system running only receivers, or running any combination of transmitters and receivers.

6.2 Usage Notes and Restrictions

6.2.1 Differences from a Conventional UART

- The UART function does not implement MODEM control signals like RTS, CTS, and CD.
- The receiver does not provide an overrun bit that is set when a received data word is not read by the CPU before a new data word arrives.
- The transmitter does not provide an underrun bit.
- The status bits are not cleared automatically by reading or writing the data registers.
- The number of stop bits is fixed to one.

6.2.2 Restrictions

To minimize TPU loading, the receiver does single sampling only. Each bit is sampled only once in the middle of the bit time - any glitch on the receive data line may cause erroneous data. The receiver does not detect idle line or break conditions, nor does the transmitter generate a break character.

Status bits must be handled by the CPU. These bits are set automatically by the TPU, but cannot be cleared by the function. The bits must be cleared by the CPU. This may cause a problem when the interrupt status bit is used to indicate that data has been received.

The interrupt status bit must be cleared by the CPU immediately before and after a read. The problem arises if the UART function receives new data before both actions are complete.

If the status bit is cleared immediately before reading data, a new data word might arrive before the previous data is read. In this case, the new word would be read, then the status bit would be set again, causing the word to be read a second time.

If the status bit is cleared immediately after data is read, a new data word might arrive before the status bit is cleared. In this case, the new data is not read, because the interrupt status bit is not set again.

To avoid these problems, the read routine must respond to the interrupt status bit quickly. The routine must execute completely before the function copies the received value from SHIFT_REGISTER to RECEIVE_DATA_REG.

6.3 Noise Immunity

The UART function is designed to filter out individual pulses which are too short to be measured correctly. These will not cause anomalous results in any of the measurement modes. However, repetitive noise on the input signal can cause anomalous results and also increased TPU activity, leading to an overall reduction in system performance. For this reason, every effort should be made to present the TPU with a noise free signal. Guaranteed minimum measurable pulse width or period can be determined by calculating worst-case latency for the UART function. Refer to Section 4, “Function State Timing” for more information.

THIS PAGE INTENTIONALLY LEFT BLANK

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

freescale™
semiconductor

AN2371/D