Using Serial Port 0 of the ADMC300 as a UART Interface

AN300-08

Table of Contents

Summary

This application note describes how to set up the serial port of the ADMC300 DSP-based motor controller for a universal asynchronous receiver/transmitter (UART) interface. This interface is widely used for instance for serial communication with PCs. The software described herein contains a set of subroutines for sending and receiving data, which permits easy usage in user applications, as will be shown in an example.

1 Principle of operation

1.1 Introduction

The Universal Asynchronous Receiver/Transmitter (UART) controller is the key component of the serial communications subsystem of a computer. The UART takes bytes of data and transmits the individual bits in a sequential fashion. At the destination, a second UART re-assembles the bits into complete bytes.

Asynchronous transmission allows data to be transmitted without the sender having to send a clock signal to the receiver. Instead, the sender and receiver must agree on timing parameters in advance and special bits are added to each word, which are used to synchronise the sending and receiving units.

When a word is given to the UART for asynchronous transmissions, a bit called the "Start Bit" is added to the beginning of each word that is to be transmitted. The Start Bit is used to alert the receiver that a word of data is about to be sent, and to force the clock in the receiver into synchronisation with the clock in the transmitter. After the Start Bit, the individual bits of the word of data are sent, with the Least Significant Bit (LSB) being sent first. Each bit in the transmission is transmitted for exactly the same amount of time as all of the other bits, and the receiver "looks" at the wire at approximately halfway through the period assigned to each bit to determine if the bit is a "1" or a "0".

The sender does not know when the receiver has "looked" at the value of the bit. The sender only knows when the clock says to begin transmitting the next bit of the word.

When the entire data word has been sent, the transmitter may add a so-called Parity Bit, which may be used by the receiver to perform simple error checking. However, this feature is not used in this example. Then, at least one Stop Bit is sent by the transmitter. If the Stop Bit does not appear when it is supposed to, the UART considers the entire word to be garbled and will report a Framing Error to the host processor when the data word is read. Regardless of whether the data was received correctly or not, the UART automatically discards the Start and Stop bits.

If another word is ready for transmission, the Start Bit for the new word can be sent as soon as the Stop Bit for the previous word has been sent. Because asynchronous data is "self synchronising", if there is no data to transmit, the transmission line can be idle.

1.2 UART communication on the serial port of the ADMC300

Because the serial ports of the ADMC DSP controller are inherently synchronised by a clock signal (SCLK), the UART interface has to be emulated by software. This is achieved as follows: Given the baud-rate of the desired UART communication, the serial clock is run at three times this value. That entails that for each bit transmitted from the host, the ADMC will read three (ideally identical) bits. The software then provides for extracting the middle of the three, which is considered to be the correct value, and to assemble the extracted bits into a byte of data. Similarly, for each bit that is to be sent to the host, the ADMC is required to repeat its value for three SCLK cycles. The protocol that is used in this example requires one start bit, followed by eight data bits (one byte) and one stop bit. No parity bit is used. That means that for each data byte, the serial port will send or receive 30 bits, as depicted in Figure 1. The routines described herein provide for converting a byte into this format and vice-versa.

Data format: UART	Start-Bit	Data Byte = 8 bits								Stop-Bit
		LSB	1	2	3	4	5	6	MSB	
Equivalent bit-pattern for SPORT0	000	xxx	yyy	xxx	yyy	xxx	yyy	xxx	yyy	111
	3 zeros	Byte represented by 24 bits								3 ones

Figure 1: TOP row - Data format for UART (1 start-, 1 stop-bit, no parity) , BOTTOM row – equivalent format for SPORT0 operation

1.3 Serial port configuration

Since each byte of data to be exchanged on the serial interface actually requires 30 bits to be transmitted or received, it has to be split in the SPORT registers into two halves of 15 bits. The word-length SLEN is therefore set to 14[1]. The data is right justified into bits 0 to 14 of the transmit and receive buffer registers. The recommended configuration for the transmitter is alternate framing mode, internally generated and active low frame sync signal. The serial clock has of course to be generated internally, too. The receiver is set up differently for each half. The first half is received in unframed mode (alternate framing). For the second word, an external frame sync is required (active high). More details about this may be found in section 4.1.

1.4 Hardware requirements

The serial interface consists of the transmitter data line (pin DT0 of the ADMC300) and the receiver data line (pin DR0 on the ADMC300). In order to generate interrupts, the receiving data line DR0 must be tied also to the RFS0 pin of the ADMC300.

If shifting from the TTL level (0V and +5V) to UART levels (-10V and +10V) is required (for instance, when communicating with a PC), some additional hardware is necessary. The schematic shown in Figure 2 makes use of the AD7306 converter. Also, the DSP side is isolated from the other device by means of standard opto-couplers such as the HCPL0630.

Of course, the AD7306 is only needed if RS232 voltage levels are required. The isolation is also optional. The routines presented here could be used to implement an SCI/UART type interface between processors at TTL voltage levels. In that case, only pull-up resistors on DT0 and DR0 (RSF0 still has to be tied to DR0) are required.

[1] Refer to ADSP-2100 Family User's Manual, chapter 5

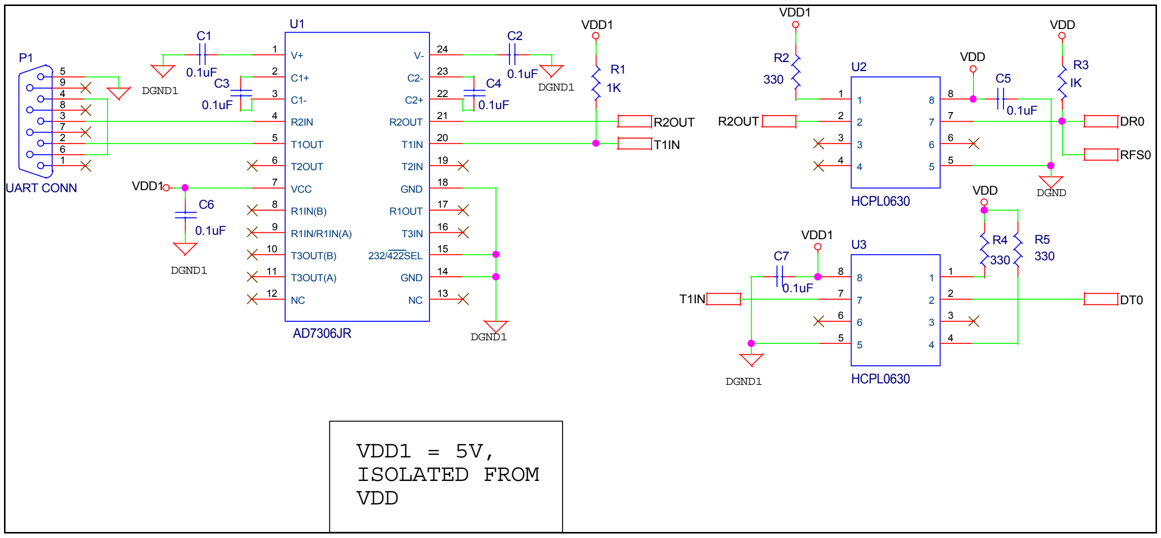

Figure 2: Recommended UART interface with level shifting and isolation

2 The UART Library Routines

2.1 Using the UART routines

The UART routines are developed as an easy-to-use library, which has to be linked to the user's application. The library consists of two files. The file "uart0.dsp" contains the assembly code for the required subroutines. This package has to be compiled and can then be linked to an application. The user simply has to include the header file "uart0.h", which provides function-like calls to the routines.

The routines require some configuration constants, which are declared in a dedicated section of the main include-file "main.h" that comes with every application note. For more information about the general structure of the application notes and including libraries into user applications refer to the Library Documentation File. Section 3 shows an example of usage of this library.

In the following sections each routine is explained in detail with the relevant segments of code which is found in either "uart0.h" or "uart0.dsp". For more information (e.g. about register use) see the comments in those files.

The following table summarises the set of macros defined in this library.

Operation	***Usage***
Initialisation	UART0_Init;
Read Byte from Host	UART0_Read(register);
Write Byte to Host	UART0_Write(register or constant);

Table 1 Command overview

There is an initialisation routine UART0_Init that has to be invoked prior to using the interface. After that data bytes may be sent or received from the host by calling the routines UART0_Write or UART0_Read, respectively.

2.2 The UART interface

The library may be accessed by including the header file "uart0.h" in the application code, as was already explained in the previous section.

It expects some constants defined in "main.h". These are two parameters that the user may want to modify, namely namely the frequency of the crystal that is used on the board (CLKIN; defaults to the value on the evaluation board of the ADMC300) and the desires transfer speed. The constant **UART0_Baudrate** defines the latter. Common values for this parameter are 1200, 2400,…, 115200 [baud/sec]. The configuration values of SPORT0 are then derived from these constants.

```
{***************************************************************************
*                                                                          *
* Constants that need to be defined in main.h:                             *
*                                                                          *
* .CONST  Cry_clock        = xxxx;        Crystal clock frequency [kHz]    *
* .CONST  UART0_Baudrate   = xxxx;        Desired Baudrate [Baud]          *
***************************************************************************}
```

The header file gives external access to the UART-routines. It is mostly self-explaining. Note how the last two macros are used to pass a parameter.

```
.MACRO UART0_Init;
    call UART0_Init_;
.ENDMACRO;

.MACRO UART0_Write(%0);
    AX0=%0;
    call UART0_Write_;
.ENDMACRO;

.MACRO UART0_Read(%0);
    call UART0_Read_;
    %0=AX0;
.ENDMACRO;
```

2.3 Implementation of the UART routines

In the following a more detailed description of the interface code included in the file "uart0.dsp" will be given. The next sections will describe the initialisation (including definitions and declarations), the receiver- and the transmitter-routines separately.

2.3.1 The initialisation of SPORT0: UART0_Init_

The clock speed of the SPORT is calculated from the configuration parameters. Next, some variables are declared which are used locally in this module. Their use will be described in more detail in the following sections. Note that there is a variable called UART0_Status, which represents the UART status at every time. This is achieved through six flag bits in the variable. They are defined after the declaration section.

```
{***************************************************************************
* Calculate Configuration Register Contents from Parameters                *
***************************************************************************}

.CONST    UART0_Clk = Cry_clock*1000/3/UART0_Baudrate;   { Ratio between Crystal frequency
                                                           and frequency of the serial
                                                           port clock  }
{***************************************************************************
* Local Variables Defined in this Module                                   *
***************************************************************************}

.VAR/RAM/DM/SEG=USER_DM      RX0_DATA;
.VAR/RAM/DM/SEG=USER_DM      TX0_DATA;
.VAR/RAM/DM/SEG=USER_DM      RX0A;
.VAR/RAM/DM/SEG=USER_DM      RX0B;
.VAR/RAM/DM/SEG=USER_DM      TX0A;
.VAR/RAM/DM/SEG=USER_DM      TX0B;
.VAR/RAM/DM/SEG=USER_DM      UART0_Status;
```

```
.VAR/RAM/DM/SEG=USER_DM      TX0_Isr_context_backup;     {only ar needs to be saved }
.VAR/RAM/DM/SEG=USER_DM      RX0_Isr_context_backup;     {only ar needs to be saved }

{*****************************************************************************************
* Local #defines in this Module                                                          *
*****************************************************************************************}

#define UART0_FirstWordReceived           0
#define UART0_SecondWordReceived          1
#define UART0_ReceiveBufferFull           2

#define UART0_FirstWordTransmitted        4
#define UART0_SecondWordTransmitted       5
#define UART0_TransmitBufferNotEmpty      6
```

Now the actual assembly code may be analysed. The ***UART0_Init_*** *subroutine initialises the SPORT0 registers with the appropriate constants for the mode described in section 1.3, set for the reception of a first half. Then the status flags are all reset, indicating that there are no pending transmissions and no data has been received. At last, the routine enables the serial port after clearing any pending (and not desired) interrupt. A call to this routine is necessary for every application before making use of the UART interface.*

```
UART0_Init_:

     .CONST  UART0_Configuration    = b#0101111001001110;
     { Configuration: Internally generated serial clock : ISCLK =1
                    Receive frame sync NOT required   :  RFSR  =0
                    Receive alternate framing mode    :  RFSW  =1
                    Transmit frame sync required    :    TFSR  =1
                    Transmit alternate framing mode   :  TFSW  =1
                    Internal transmit frame sync ON   :  ITFS  =1
                    Internal receive frame sync OFF   :  IRFS  =0
                    Internal receive frame sync OFF   :  IRFS  =0
                    Internal receive frame sync OFF   :  IRFS  =0
                    Transmit frame sync active LOW          :    INVTFS=1
                    Receive frame sync active HIGH   :   INVRFS=0
                    Data format right just., 0 filled :  DTYPE =b#00
                    Word length 15 bits             :    SLEN  =0xe     }

      Clear_Bit_DM(sport0_ctrl_reg, 14);
      Write_DM(sport0_sclkdiv, UART0_Clk - 1);     {sets serial clock frequency}
     Write_DM(sport0_ctrl_reg, UART0_Configuration);

      Write_DM(UART0_Status, 0x0000);      {clear all status flags               }

     Set_InterruptVector(RX0_INT_ADDR, UART0_RX_Isr); { Vector for UART0_RX_Isr    }
     Set_InterruptVector(TX0_INT_ADDR, UART0_TX_Isr); { Vector for UART0_TX_Isr    }

     ifc = 0x0060;                 {clear pending SPORT0 interrupts     }
     ar = IMASK;
     ar = setbit 5 of ar;          {enable receive and ...                     }
     ar = setbit 6 of ar;          {... transmit interrupts for SPORT0   }
     IMASK = ar;

      Set_Bit_DM(SYSCNTL, 12);      {enable SPORT0                      }

     rts;
```

2.3.2 The transmitter routine: UART0_Write_

The transmitter routine makes use of three of the status flags in order to ensure the correct sequence of the operations. The flag **UART0_TransmitBufferNotEmpty** indicates that a previous transmission is still in progress when it is set. It is initially polled until there are no pending transmissions, i.e. the buffer is empty. At this point the demanded transmission may be initiated. The above mentioned flag is set, and two other bits, namely **UART0_FirstWordTransmitted** and **UART0_SecondWordTransmitted** are cleared to indicate that none of the two words has been transmitted yet. The data byte in ax0 is processed to generate the two 15-bit words to be sent. The next action that is required is to send the first word and update the status flag **UART0_FirstWordTransmitted** by setting it, which indicates that the first word has been transmitted. For correct operation under all conditions, these operations may not be interrupted by any other service routine. This is ensured by the adopted sequence of disabling any interrupt, performing the two actions and enabling interrupts after that again. The SPORT will generate an interrupt as soon as it is ready to transmit another word. This will call the interrupt service routine **UART0_TX_Isr**, which provides for sending the 2nd word after checking the status. Again, updating the status and sending the word is not interruptible by means of the sequence described above. This time the flag **UART0_SecondWordTransmitted** is set. The SPORT will generate another interrupt when ready for the next transmission. Now, since the status indicates that both halves are sent, no further transfer is required and the **UART0_TransmitBufferNotEmpty** is cleared to indicate that the next transmission may be started.

With the general description above the following two pieces of code should be mostly self-explaining. The routine UART0_Write_ is the, globally accessible, entry point, which initiates the transmission of the data contained in the low-byte of ax0. The corresponding code is shown in the box below.

```
UART0_Write_:

            ar=dm(UART0_Status);
            ar=TSTBIT UART0_TransmitBufferNotEmpty of ar;
            if ne jump UART0_Write_;{ Wait until previous transfer is completed }

            ar=dm(UART0_Status);
            ar=CLRBIT UART0_FirstWordTransmitted of ar;
            ar=CLRBIT UART0_SecondWordTransmitted of ar;
                                  { None of the 30 Bits has been transmitted yet       }
            ar=SETBIT UART0_TransmitBufferNotEmpty of ar;
                                  { Indicate that Transmission is in progress  }
            dm(UART0_Status)=ar;

            call PROCESS_TX0;     { Build the two words to be sent from the data  }

            ar=dm(UART0_Status);
            ar=SETBIT UART0_FirstWordTransmitted of ar;
            dis INTS;             { following two instructions uninterruptible }
            dm(UART0_Status)=ar;  { update status and                          }
            TX0=dm(TX0A);         { send the 1st half (the 2nd is handled by   }
            ena INTS;             { the interrupt service routine)             }

            rts;
```

The interrupt service routine is reported hereafter. It is worth to note, that ar is the only register that is used by this routine. It has been decided therefore to save it explicitly into a backup variable and restore it before exiting the routine instead of switching to the secondary register set of the ADMC300. This leaves the option of nesting interrupts without having to worry about register alterations in another service routine written by the final user.

```
UART0_TX_Isr:

            DM(TX0_Isr_context_backup) = ar;     {save context                 }

            ar = dm(UART0_Status);
            ar = TSTBIT UART0_TransmitBufferNotEmpty of ar;
            if eq jump Exit_TX0_Isr;      { If the buffer is empty do nothing   }

            ar = dm(UART0_Status);
            ar = TSTBIT UART0_SecondWordTransmitted of ar;
            if eq jump SendSecondWord;    { If the First word has been transmitted
                                            send the second}

            ar = dm(UART0_Status);
            ar = CLRBIT UART0_TransmitBufferNotEmpty of ar;
            dm(UART0_Status)=ar;          { If the second word has been transmitted
                                            reset status }

            jump Exit_TX0_Isr;
```

```
SendSecondWord:

            ar = dm(UART0_Status);
            ar=SETBIT UART0_SecondWordTransmitted of ar;
            dis INTS;              { following two instructions uninterruptible }
            dm(UART0_Status)=ar;   { update status and                          }
            TX0=dm(TX0B);          { send the 2nd half                          }
            ena INTS;

Exit_TX0_Isr:
            ar = DM(TX0_Isr_context_backup);      {restore context            }
            rti;
```

The remaining part is the actual conversion from the data-byte to the 30-bit word to be transmitted. The word is built in the shift register sr1 and sr0. First, the stop bit (represented by the binary sequence 111) is inserted into positions 25-27. Then, for each bit to be inserted, a corresponding sequence of 000 or 111 will be ORed into the register at positions 28-30. Simultaneously, the whole register is shifted down by three positions, so the next bit may be inserted in the same positions as the previous one. After the loop, the shift register is made up, from the right to the left, of the stop bit, the eight data bits (represented by 24 bits), beginning with the most significant bit, and the start bit (000, inserted automatically by the shift operations). These 30 bits occupy positions 1 to 30 of the register. The reason for this is that, since the SPORT requires the 15 bits to be aligned to the right, the upper half of the shifter contains the first word to be transmitted already aligned properly, and the lower half needs only to be shifted by one position to the right for correct alignment. The last instructions do this and store the register contents into a buffer variable TX0A and TX0B in data memory.

```
PROCESS_TX0:

            dm(TX0_DATA)=ax0;              { save content of ax0         }
            ar =ax0;                       { ar=byte to be transmitted   }
            sr = LSHIFT ar by 8 (LO);      { shift data byte into high-byte}
            ax0= sr0;

            sr1=0x0e00;                    {stop bit in position 25-27, will
                                            be shifted down when inserting data bits}
            sr0=0x0;

            cntr = 8;                      {insert 8 bits into sr        }
            do Insert_Bit until CE;
              ay0=ax0;
              ar = ax0 + ay0 ;        {left shift 1 bit, highest bit into carry flag }
              ax0= ar;                {store shifted value in ax0 }
              ar=sr1;                 {store higher word in ar, preserve previous status }
              if NOT AC jump Bit_0;          {test bit shifted into carry                  }
Bit_1:                ay0=0x7000;          {if 1, expand to 111(aligned to positions 28-30) }
              ar= ar or ay0;               {ar=111 ored into current higher word }
Bit_0:             sr= LSHIFT sr0 by -3 (LO);      {shift 32 bit word down three positions }
Insert_Bit: sr=sr or LSHIFT ar by -3 (HI);         {for next bit and insert current bit  }

                          { after 8 bits, the word is aligned to bits 1-30 of sr  }
                          { start bit is automatically in positions 28-30 (000)       }

            dm(TX0A)=sr1;            {TX0A gets 15 MSBs = highword of sr       }
            sr= LSHIFT sr0 by -1 (LO);
            dm(TX0B)=sr0;            {TX0B gets 15 LSBs from sr0 aligned to the right }

            ax0=DM(TX0_DATA);                     {restore content of ax0              }
            rts;
```

2.3.3 The receiver routine: UART0_Read_

The transmitter routine makes use of three of the status flags in order to ensure the correct sequence of the operations. The flag **UART0_ReceiveBufferFull** indicates that a complete 30-bit word has been received and is ready to being processed, when it is set. It is initially polled until a word is received, i.e. the buffer is full. At this point the data byte is extracted from the 30-bit word and stored into ax0. The status is reset to indicate that the routine is ready for a new operation. Unlike the transmitter, the receiving routine has no control over the start of the data transfer. When a word is received the SPORT generates an interrupt. Depending on the status of the flags, namely **UART0_FirstWordReceived** and **UART0_SecondWordReceived**, the service routine undertakes different actions. When **UART0_FirstWordReceived** is cleared, indicating that none of the two words has been received yet, the word is stored in RX0A, SPORT0 is set up in the receiving mode for the second half and the status is updated. If the flag **UART0_SecondWordReceived** is cleared, the word goes to the second half RX0B, SPORT0 is set up in the receiving mode for the next first half and the status is updated. If none of the above conditions is true then the currently received word was received before Read_UART0 has processed the previous words. In that case, the word is removed from the SPORT0 receive register and is lost for any further operation.

With the general description above the following two pieces of code should be mostly self-explaining. The routine UART0_Read_ is the, globally accessible, entry point, which waits for data to be received over SPORT0. It then extracts the data-byte and stores the in the low-byte of ax0. The corresponding code is shown in the box below.

```
UART0_Read_:

            ar=dm(UART0_Status);
            ar=TSTBIT UART0_ReceiveBufferFull of ar;
            if eq jump UART0_Read_;       { wait until complete byte has been received }

            call PROCESS_RX0;      { extract byte from the 30 bit word          }

            ar=dm(UART0_Status);
            ar=CLRBIT UART0_FirstWordReceived of ar;
            ar=CLRBIT UART0_SecondWordReceived of ar;
            ar=CLRBIT UART0_ReceiveBufferFull of ar;
            dm(UART0_Status)=ar;   { restore status to ready for new reception  }

            ax0=dm(RX0_DATA);       { store data into ax0                              }

            rts;
```

The interrupt service routine is reported hereafter. As for the transmitter service routine, ar is the only register that is used by this routine. Again and for the same reason, it is saved explicitly into a backup variable and restored before exiting the routine instead of switching to the secondary register set of the ADMC300.

```
UART0_RX_Isr:

            DM(RX0_Isr_context_backup) = ar;      { save context                }

            ar = dm(UART0_Status);
            ar = TSTBIT UART0_FirstWordReceived of ar;
            if ne Jump SecondWord;        { first word was already received    }

            ar=dm(Sport0_Ctrl_Reg);
            ar=SETBIT 13 of ar;
            dm(Sport0_Ctrl_Reg)=ar;       { set SPORT receive mode for 2nd half }

            dm(RX0A)=RX0;                 { store first word                   }

            ar = dm(UART0_Status);
            ar=SETBIT UART0_FirstWordReceived of ar;
            dm(UART0_Status)=ar;          { update status flags                }

            jump Exit_RX0_Isr;

SecondWord:
            ar = dm(UART0_Status);
            ar = TSTBIT UART0_SecondWordReceived of ar;
            if ne Jump FlushWord;
                                          { this is second half                }
            ar=dm(Sport0_Ctrl_Reg);
            ar=CLRBIT 13 of ar;
            dm(Sport0_Ctrl_Reg)=ar;       { prepare SPORT0 for the next 1st half       }
            DM(RX0B)=RX0;                 { store second word                  }

            ar = dm(UART0_Status);
            ar = SETBIT UART0_SecondWordReceived of ar;
            ar = SETBIT UART0_ReceiveBufferFull of ar;
            dm(UART0_Status)=ar;          { indicate that complete byte has
                                            been received       }

            jump Exit_RX0_Isr;

FlushWord:
            ar=RX0;

Exit_RX0_Isr:
            ar = DM(RX0_Isr_context_backup);      { restore context             }
            rti;
```

The remaining part is the actual conversion from the 30-bit word to the data-byte. The received word is aligned to bits 0 to 22 of the shifter (the stop bit and 1 bit is shifted out at the right side in order to align the middle bit of the first bit). Then, in a loop, each bit is read from position 0, inserted into ar and the word shifted to the right by three positions. Therefore the data-byte is built in ar from the middle of each triple of bits.

```
PROCESS_RX0:

            mr1=DM(RX0A);                   {1st half received in RX0A          }
            mr0=DM(RX0B);                   {2nd half received in RX0B          }

            sr= LSHIFT mr0 by -4 (LO);      { shift 3 stop bits + 1 bit out             }
                                           { to get to middle of 1st bit               }
            sr=sr or LSHIFT mr1 by -5 (HI); { MSW shifted down and or'd in              }

            ay1=0x0;                        { ay1 contains final byte           }

            ay0=0x1;
            CNTR = 0x7;                   { extract seven bits                 }
            do Extract_Bit until CE;
                   ar= sr0 and ay0 ;       { extract bit                        }
                   ar= ar or ay1 ;         { OR into byte                               }
                   ay1=ar;                 { also create mirror image           }
                   ar= ar + ay1 ;          { shift it left 1                    }
                   ay1=ar;
                   mr0=sr1;
                   sr= LSHIFT sr0 by -3 (LO);      { shift 30 bit word down to }
Extract_Bit:       sr=sr or LSHIFT mr0 by -3 (HI);         { next bit                   }

            ar= sr0 and ay0;                { extract 8th bit                   }
            ar= ar or ay1;                  { final byte created                }
            DM(RX0_DATA)=ar;                { store in RX0_DATA buffer          }

            rts;
```

3 Usage of the library: an example

3.1 The main program: main.dsp

The example demonstrates how to set up a communication between a host PC and the ADMC300, while simultaneously generating a three-phase sine wave by means of the PWM block. The application has already been described in a previous note[2]. This section will only explain the few and intuitive additions to be made to that application. The duty-cycle values are generated within an interrupt service routine.

[2] AN300-03: Generation of Three-Phase Sine-Wave PWM

The main loop is used for processing received bytes of data, echoing them back and increase or decrease the frequency of the sine-wave whenever a '+' or a '-' is received.

The file "main.dsp" contains the initialisation and PWM Sync and Trip interrupt service routines. To activate, build the executable file using the attached **build.bat** either within your DOS prompt or clicking on it from Windows Explorer. This will create the object files and the **main.exe** example file. This file may be run on the Motion Control Debugger.

In the following, a brief description of the additional code (put in evidence by bold characters) is given.

Start of code – declaring start location in program memory

```
.MODULE/RAM/SEG=USER_PM1/ABS=0x60     Main_Program;
```

Next, the general systems constants, the PWM configuration constants and the UART configuration parameters (main.h – see the next section) are included. Also included are the PWM library, the trigonometric library and the ***UART interface.***

```
{*********************************************************************************
* Include General System Parameters and Libraries                               *
*********************************************************************************}
#include <main.h>;
#include <pwm300.h>;
#include <trigono.h>;
#include <uart0.h>;
```

Variables, Labels and Scope definitions are identical to AN300_03.

```
{
            ... omissis ...
}
```

Note the initialisation of the UART interface.

```
Startup:

     UART0_Init;

     PWM_Init(PWMSYNC_ISR, PWMTRIP_ISR);
     IFC = 0x80;                  { Clear any pending IRQ2 inter.      }
     ay0 = 0x200;                 { unmask irq2 interrupts.     }
     ar = IMASK;
     ar = ar or ay0;
     IMASK = ar;                  { IRQ2 ints fully enabled here       }
```

The main loop: Instead of just waiting for PWM interrupts, this time the main loop executes the routine for receiving a data byte. Read_UART0 polls a status bit until a byte is received from the host. Nevertheless, the PWM interrupt service routine is still executed with the desired frequency. As soon as the byte is received, it is echoed back with a call to Write_UART0. After that, its value is compared with the ASCII codes for '+' and '-' respectively. If it is equal to one of them, the reference value AD_IN is modified by an increment of +0x100 or –0x100. Then the cycle begins again.

```
Main:                       { Wait for interrupt to occur, meanwhile handle UART communication}
     UART0_Read(AR);
     UART0_Write(AR);

     ay1= 0x100;

     ay0=43;        {ascii code for '+'}
     ar=ax0-ay0;
     if ne jump Comp_minus;
     ar = DM(AD_IN);
     ar = ar +ay1;
     DM(AD_IN)=ar;
     jump MAIN;
Comp_minus:
     ay0=45;        {ascii code for '-'}
     ar=ax0-ay0;
     if ne jump MAIN;
     ar = DM(AD_IN);
     ar = ar -ay1;
     DM(AD_IN)=ar;
     jump Main;
```

```
RTS;
```

The interrupt service routine generates the duty-cycle values for the PWM block. This routine makes use of the secondary register set of the ADMC core. This allows for single cycle context switching, in order to not interfere with the main loop actions. Refer to the cited application note for more details on it.

```
{***********************************************************************************************}
{ PWM Interrupt Service Routine                                                         }
{***********************************************************************************************}

PWMSYNC_ISR:
            ena sec_reg;

            my0 = DM(AD_IN);

            mr = 0;                               {Clear mr                            }
            mr1 = dm(Theta);                       {Preload Theta                       }
             mx0 = Delta;
            mr = mr + mx0*my0 (SS);               {Compute new angle & store           }
             dm(Theta) = mr1;
            Sin(mr1);                             { Result in ar register              }
             mr = ar*my0 (SS);                    { Multiply by Scale for VrefA        }
            dm(VrefA) = mr1;

             ax1 = dm(Theta);                     { Compute angle of phase B           }
            ay1 = TwoPioverThree;
             ar = ax1 - ay1;
            Sin(ar);                              { Result in ar register              }
             mr = ar*my0 (SS);                    { Multiply by Scale for VrefA        }
            dm(VrefB) = mr1;

             ax1 = dm(Theta);                     { Compute angle of phase C           }
             ay1 = TwoPioverThree;
             ar = ax1 + ay1;
            Sin(ar);                              { Result in ar register              }
             mr = ar*my0 (SS);                    { Multiply by Scale for VrefA        }
             dm(VrefC) = mr1;

                ax0 = DM(VrefA); ax1 = DM(VrefB); ay0 = DM(VrefC); ay1= DM(Theta);
            PWM_update_demanded_Voltage(ax0,ax1,ay0);

RTI;
```

3.2 The main include file: main.h

This file contains the definitions of ADMC300 constants, general-purpose macros and the configuration parameters of the system and library routines. It should be included in every application. For more information refer to the Library Documentation File.

This file is mostly self-explaining. The relevant section to this example is shown here. As described in the Library Documentation File, every library routine has a section in "main.h" for its configuration parameters. The following defines the parameters for the UART interface used in this example. The baudrate is expressed in Baud.

```
{*********************************************************************************************}
{ Library: UART0                                                                              }
{ file   : UART0.dsp                                                                          }
{ Application Note: Using the serial port as a UART                                           }

.CONST UART0_Baudrate =115200;                 {Desired Baudrate [Baud]                      }
{*********************************************************************************************}
```

4 Additional comments

4.1 Receiver configuration

As mentioned in section 1.3, the interface requires a different SPORT configuration for each halve of the received word. The reason for this is explained by means of a measured timing diagram, such as shown in Figure 3. The represented relation between serial clock and data receive line is only indicative since data is transmitted asynchronously from the host.

The receiver is initially set up with the RFSR (receive frame sync required) bit equal to zero. This entails that the frame sync signal (tied to data receive line) is required only at the beginning of the data acquisition. Therefore, a continuous stream of data would be read in. When the first halve is read, SPORT0 generates an interrupt (the first glitch of the bottom track in the figure below) and starts reading the next bits. The interrupt routine now changes the configuration by setting the RFSR bit. SPORT0 will from now on only acquire data if a new frame sync signal is applied. However, the current word (the 2nd halve) is still read. When this word is completely transferred, a second interrupt is generated and the service routine resets the RFSR bit to the initial condition. The UART specification requires the DR0 line to be held high. SPORT0 will only recognise a new frame sync signal when is goes low again, which is when the start-bit of the next data is sent. Refer to figures 5.13 and 5.15 of the ADSP 2100 family user's manual for more information.

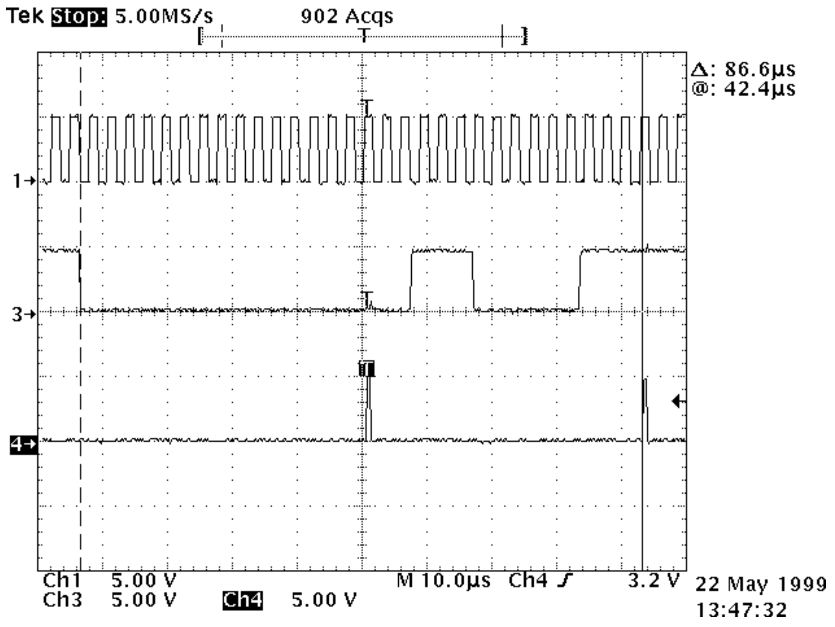

Figure 3: Typical timing diagram during receive operation. Top: serial clock SCLK0; MID: data receive DR0; BOTTOM: execution of the interrupt service routine UART0_RX_Isr.

4.2 Reliability of the communication

Since the process of receiving is, due to the nature of the SPORTs, trying to synchronise an inherently asynchronous signal to the internal serial clock, certain misinterpretations cannot be avoided with these simple, three-times oversampling, routines. The falling edge of the start-bit may fall anywhere within one cycle of the serial clock. The receiver then stores 30 bits on the falling edges of the serial clock. However, if the falling edge of the start-bit coincides with the falling edge of the clock, that bit may be erroneously read in and the whole word is anticipated by one bit. This may lead to incorrect interpretation. Tests have shown that this occurs with a probability of approx. 80ppm, when data is continuously sent. The probability decreases drastically if delays are inserted between the single bytes to be transmitted.

In order to avoid unacceptable behaviours, it is recommended to echo the received data to the host, which should check for eventual misinterpretation.

If a more robust transmission is required, the three-times sampling is not suited. One may think of increasing the sampling factor to 5 or 6. This entails more complicate routines since more words have to be read and sent for each byte, but on the other hand allow for error detection and correction by testing more samples for each bit and implementing some kind of majority tests.

4.3 Interrupt configuration and context swapping

As mentioned in sections 2.3.2 and 2.3.3, it has been decided to use the primary set of registers in the interrupt service routines and to manually save the context rather than swapping to the secondary set provides with the ADSP-21xx core. The reason for it is that serial communication tasks are normally strongly linked to main loop routines. Since there is only one register that is required by the service routines, the overhead created by the context saving instruction is not critical. The advantage of this approach is that the option of nesting interrupts, such as PWM service routines with the serial communication routine is left to the user of this interface. As shown in the example, the PWM routine swaps the set of registers, and may be nested with the serial interrupts with no additional code.