Objectives

- To establish bidirectional communications with an external host computer via the serial interface of the microcontroller
- To write a set of functions for the input and output of clear text and numerical data from and to a terminal program

Introduction

Serial communication is central to many embedded computer systems. Being able to communicate with a microcontroller through its Serial Communication Interface (SCI) provides the developer with ‘eyes’ to see what is going on inside the chip. When working with a new microcontroller, one of the first tasks commonly is the writing of a few communication routines through which error messages and status information can be displayed and control signals can be passed to the program.

The MC9S12DP256B/C comes with two independent serial interface units (SCI0 and SCI1). A small memory resident monitor program has been installed on all Dragon12 boards in our laboratory. This program is started upon a hardware reset. It uses interface SCI0 to communicate with an external debugger on the host, e.g. the Hi-Wave source level debugger which has been integrated into the CodeWarrior IDE. The monitor/debugger facilitates the download of user programs into RAM or Flash EEPROM, as well as their on-chip debugging using hardware breakpoints; hardware breakpoints allow a program to be interrupted, even if it has been installed in ROM. Interface SCI0 is configured for a line speed of 115200 bps, with 8 data bits, no parity checking and one stop bit (115200-8/N-1). The second serial communication interface, SCI1, is available for general use.¹

Figure LMP6-1 shows the principal elements of an SCI unit on the MC9S12DP256B/C. The line speed is controlled by programming the baud rate generator. This unit produces an internal clock signal which determines the timing on both the transmission line (sci_tx_od) as well as the reception line (sci_rx_ind). Incoming data is stored in the Receive Shift Register before being copied to the SCI Receive Data Register. Similarly, outgoing data needs to be written to the SCI Transmit Data Register, from where it is copied to the Transmit Shift Register. Note that both the Transmit Data register as well as the Receive Data register are memory mapped at the same address. Read instructions to this address accesses the Receive Data register, whereas write instructions addresses the Transmit Data register.

All special function registers of the SCI unit are shown in Figure LM6-2. The unit offset is to be added to the base address of the selected SCI unit in the memory map of the MC9S12DP256B/C. This is commonly done within header file which defines all special function registers (here: mc9s12dp256.h).

¹ Note: Strictly speaking, both serial communication interfaces are ‘for general use’ – provided no serial port based debugging is required.
The line speed of the SCI unit is controlled through the two 8-bit SCI Baud Rate Registers, SCIBDH and SCIBDL (Figure LMP6-3). The 13-bit value \( BR = [SBR12 \ldots SBR0] \) defines a pre-scale value which is used to reduce the input clock of the SCI unit to the required line speed. This SCI input clock is \( 1/16 \)th of the central bus clock, i.e. 24 MHz / 16 = 1.5 MHz. The baud rate can thus be calculated using:

\[
 f_{\text{line}} = \frac{f_{\text{bus}}}{16 \cdot BR}
\]
Writing to the high byte of the baud rate register is ineffective until the low-byte has been written as well. Complete the following table to work out the required values for the specified baud rates (Table LMP6-1).

<table>
<thead>
<tr>
<th>Baud rate (bps)</th>
<th>BR</th>
<th>SCIxBDL</th>
<th>SCIxBDH</th>
<th>Error (rel.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1200</td>
<td>1250</td>
<td>226</td>
<td>4</td>
<td>0 %</td>
</tr>
<tr>
<td>2400</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4800</td>
<td>313 or 312</td>
<td>57 or 56</td>
<td>1</td>
<td>-0.16 % or +0.16 %</td>
</tr>
<tr>
<td>9600</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19200</td>
<td>78</td>
<td>78</td>
<td>0</td>
<td>+0.16 %</td>
</tr>
<tr>
<td>38400</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>57600</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>115200</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table LMP6-1  Baud rate generation, 24 MHz bus clock

Note that some of the above baud rates can not be produced without error. Dividing the CPU clock rate (24 MHz) by 16:19200 or 307200 yields 78.125, i.e. a fractional
number. The nearest integer number is 78. Therefore, when requesting a line speed of 19200 bps, the baud rate generator actually generates a clock signal of \(24 \times 10^6 / (16 \times 78) = 19230.77\) bps. This is a relative error of \(30.77 / 19200 = +1.6 \times 10^{-3}\) or 0.16 %. Most serial communication equipment can cope with relative baud rate errors of up to 6 %. This relative error should always be kept as small as possible.

Control registers SCIxCR1 and SCIxCR2 specify all other parameters of the interface, e.g. the number of bits per data word (8/9), the type of parity checking (even/odd/none), whether the reception/transmission complete interrupts are enabled, whether the receiver/transmitter is enabled, etc. (see: Figure LMP6-3).

![Figure LMP6-3](image)

The status registers can be used to check the current state of the interface when waiting for incoming data (Receive Data Register Full – RDRF) and/or prior to initiating a data transmission (Transmit Data Register Empty – TDRE). There are also flags that indicate the end of a transmission (Transmission Complete – TC) as well as various error conditions, e.g. when the Receive Data Register is overwritten by new incoming data before the program has managed to read the previous value (Overrun – OR), etc.

There are two SCI Data Registers, SCIxDRH and SCIxDRL. Regular 8-bit communication only makes use of SCIxDRL.

### Implementation

Write a serial communication module for SCI1 (e.g. ‘SCI1.c’), including the following functions:

```c
void SCI1_Init(unsigned long baudrate)
void SCI1_OutChar(char myChar)
char SCI1_InChar(void)
void SCI1_OutString(char *myString)
void SCI1_InString(char *myString, unsigned int maxLength)
```

All functions are to be exported as ‘external’ in a corresponding header file ‘SCI1.h’. The string functions should make call upon the character functions to transmit/receive the individual characters. Function SCI1_InString receives a pointer to a buffer of length 'maxLength'. The function should never accept more data as there is space on this buffer; all received characters are to be echoed back to the sending host (commonly a terminal program). If you are keen, you could catch non-printable characters such as ‘backspace’ and ‘delete’ or the ‘arrows’ and move the cursor accordingly.

Write a main module which calls upon the functions of your new serial communications module to read strings from a terminal connected to the serial connector ‘P2’ on the Dragon12 board. Set the communication speed to 115200 bps with 8 data bits and no parity checking. The program should then echo back whatever the user has typed on
the terminal before waiting for the next input (infinite loop). Figure LMP6-4 shows a screen shot of what the output of this program could look like.

![Figure LMP6-4](image.png) Output of the test program; SCI1 runs at 115200

**Extensions**

[1]

Write communication routines which read and write signed and unsigned decimal numbers as well as signed and unsigned hexadecimal numbers:

```
unsigned int SCI1_InUnsignedDec(void)
signed int SCI1_InSignedDec(void)
```

and

```
unsigned int SCI1_InUnsignedHex(void)
signed int SCI1_InSignedHex(void)
```

[2]

Modify ‘SCI1.c’ to implement interrupt driven background communication as shown in class. An elegant version of this program would make use of ring buffers – the macros introduced in lecture MP7 are fairly generic and should also run on the MC9S12.