Application Notes

for SystemBase's UARTs

Ver. 1.0

Serial Communication Experts
SystemBase
Since 1987

The List of Contents

AN16001. SystemBase's MIO Bus™ Overview 1

1. Overview 1

2. MIO Bus™ Main Features 1

3. MIO BusTM Signals 1

4. Octal-UART Applications using MIO Bus™ 3

AN16002. Programming Guide for SB16C105x 5

1. Introduction 5

2. Basic Techniques 5

2.1 UART Initialization 5

2.2 Interrupts 6

2.3 Registers 6

3. Programmer's Guide 6

4. Sample Code 9

4.1 Interrupt Driven Program 9

4.2 Using Global Interrupts 13

AN16003. The Benefits of UART with Deep FIFO 15

1. Overview 15

2. Deep FIFO 15

3. The Statistics of Deep FIFO Effect 16

3.1 Comparative table of the throughputs between 16B and 256B FIFO. 17

3.2 The comparative table of the CPU usage between 16-byte FIFO and 256-byte FIFO. 17

4. Conclusions 18

AN16004. Interfacing between UART and CPU 19

1. Overview 19

2. Bus Connection 19

3. Reference Schematic 21

AN16001. SystemBase's MIO Bus™ Overview

(for Octal-UART, SB16C1058)

1. Overview

SB16C1058 UART supports Normal mode in which the chip operates as other common Octal-UARTs and MIO mode which supports SystemBase's MIO mode.

SB16C1058 contains built-in control and memory circuits that allow expansion of up to 32 ports (by 8 ports) by adding MIO Bus™ Interface and Option Registers structure on a common Octal-UART. Through this method, glue-logics are unnecessary when expanding ports. With SystemBase's PCI Bridge Controller SB4002A or Generic CPU, 8, 16, 24 and 32 port serial communication PCI Card Adapter and Device Server applications can be made easily with low cost.

2. MIO Bus™ Main Features

- Expandable up to 32-port Serial Channel without any glue-logics in the MIO Bus™
- Supports flexible structures of application using the automatic daisy chain installation checking
- Fast Interrupt Service using Interrupt Vector Processing

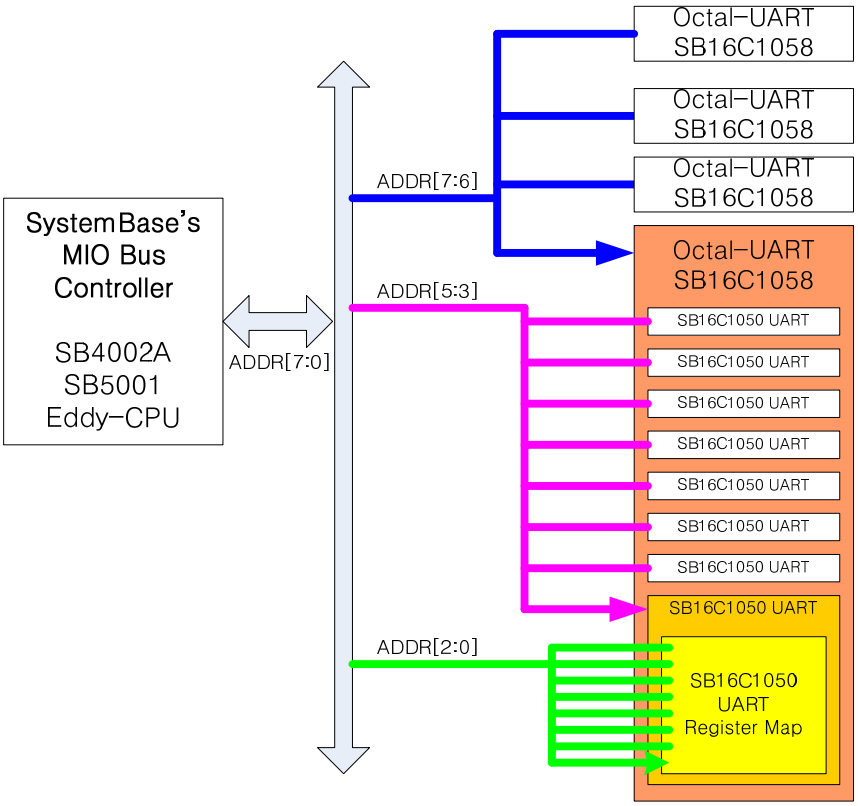
  Interrupt Vector Processing is based on Option Registers (IMR & IPR)
- S/W Notification of the serial interfaces and UART controller using Option Registers
- Support dedicated MIO chip selections, UARTn and OPTn for the efficient data transfer

3. MIO BusTM Signals

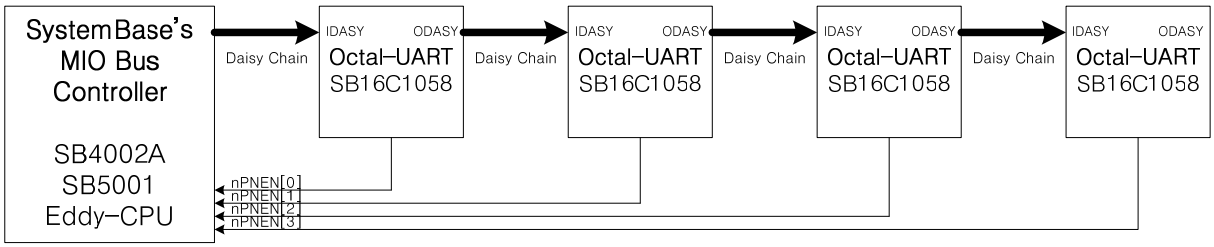
PIN Name	MIO mode	Description
ADDR[7:0]	ADDR[7:0]	MIO mode (nUART) ADDR[7:6] / ADDR[5:3] / ADDR[2:0] 4 octal-uart selection / 8 uart selection / 8 uart internal register selection MIO mode (nOPT) ADDR[5:0] : option registers selection
DATA[7:0]	DATA[7:0]	8-bits Data Bus
nIOR	nIOR	Read Strobe
nIOW	nIOW	Write Strobe
nCS_nUART	nUART	UART Registers Access Enable
nOPT	nOPT	Option Registers Access Enable
nBUF	nBUF	MIO Bus Read Enable for 245 Buffer
IDASY[1:0]	IDASY[1:0]	Input Daisy Chain for 8-port Unit Block
ODASY[1:0]	ODASY[1:0]	Output Daisy Chain for 8-port Unit Block
nPNEN0_OSC0	nPNEN0_OSC0	Input @ Initial Time

nPNEN1_OSC1	nPNEN1_OSC1	OSC[1:0] – UART operational frequency
nPNEN2_INTF0	nPNEN2_INTF0	INTF[1:0] – The kind of Serial Interfaces, RS232/422/485
nPNEN3_INTF1	nPNEN3_INTF1	Output @ Normal : nPNEN[3:0] – Octal-UART installation information

In MIO mode, ADDR[7:0] 8-bits are used. ADDR[7:6] are used to select one of 4 Octal-UARTs, ADDR[5:3] are used to select one of the 8 UART channels and ADDR[2:0] are used to select one of the selected UART channel's internal registers.

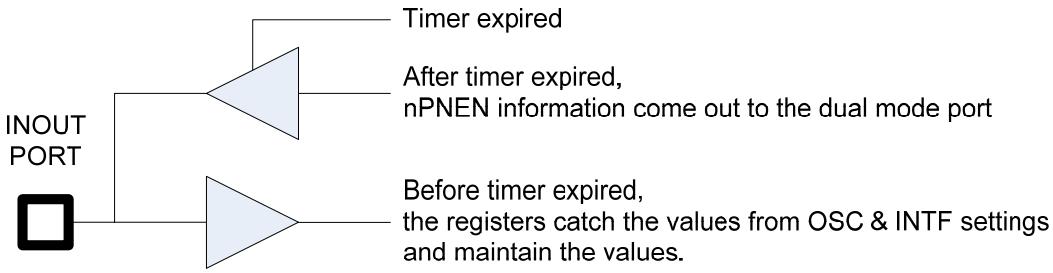


SB16C1058 can expand up to 32 ports with MIO Bus™ by 8 ports. A flexible structure can be made using MIO Bus™. Using this structure, SB16C1058 can be made into 8, 16, 24 and 32 port. In this structure, Daisy Chain is used to determine how many 8-port units are connected thereby allowing an easier management for software.



SB16C1058 has 4 Dual Mode Pins which are PNEN0n_OSC0, PNEN1n_OSC1, PNEN2n_INTF0 and PNEN3n_INTF1. These dual mode pins operate in input mode for a certain amount of time (0 ~ 4ms) after the power is inserted and work as output mode after that.

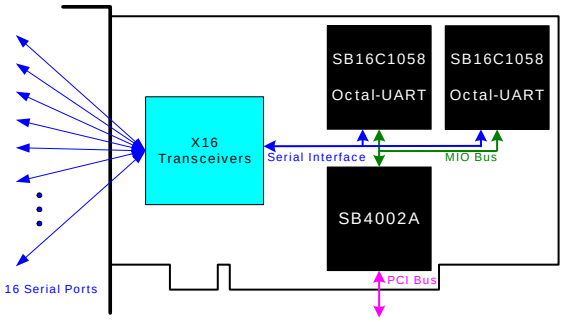
Input pin @ Initial Time	Output pin @ Normal
OSC[1:0] – UART operational frequency INTF[1:0] – The kind of Serial Interfaces	nPNEN[3:0] – Octal-UART installation information

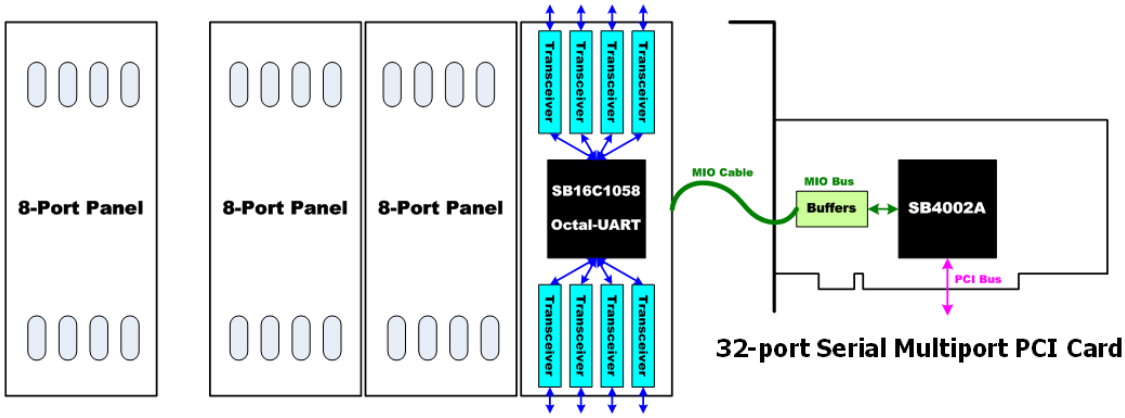


4. Octal-UART Applications using MIO Bus™

16-port serial card and expandable 32-port serial card can be made. 16-port serial card is designed to expand 16 serial ports with two SB16C1058 Octal-UARTs connected to SB4002A by MIO Bus as shown in below diagram. 32-port serial card is designed as shown in below diagram. It is composed of one SB4002A and is capable of expanding by 8 ports using 8-port panels.

When serial communication port is expanded over 8 ports, there is higher chance of the FIFO Buffer getting full in the UART as the result of PCI Bus' performance problem and as data is overwritten, overrun errors are more likely to occur. Since SB16C1058 uses 256-Byte FIFO, overrun errors and the loss of data can be prevented. Therefore, SB16C1058 can be considered the optimum solution for serial cards with multiple ports.



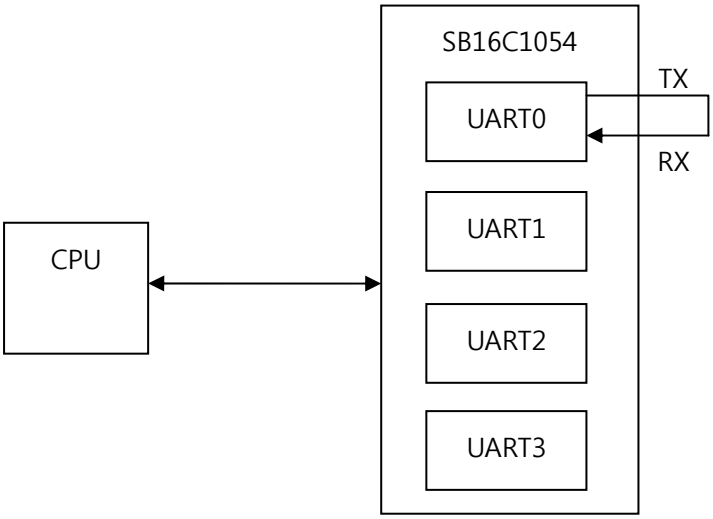

8-Port Panel
8-Port Panel
8-Port Panel
Transceiver
Transceiver
Transceiver
Transceiver
SB16C1058
Octal-UART
Transceiver
Transceiver
Transceiver
Transceiver
MIO Cable
MIO Bus
Buffers
SB4002A
PCI Bus
32-port Serial Multiport PCI Card

AN16002. Programming Guide for SB16C105x

1. Introduction

UARTs are one of simple logics of IC. Still, people who use them for the first time have difficulties in using them. This paper will provide a brief instruction for setting and testing UART assuming SystemBase SB16C1054 is used.

Usually 8051 or ARM CPU is used to control UART. We will be testing our UART by initializing the UART and transceiving data through CPU data bus. In this paper, we will use one of the four UARTs to send/receive data as below.



2. Basic Techniques

2.1 UART Initialization

1. Set LCR[7] = 1 to enable DLAB. This allows DLL, DLM Registers to be accessed.
2. Set baud rate by configuring DLL, DLM.
3. Set LCR[7] = 0, so normal registers can be accessed.
4. Set LCR to configure serial data communication frame.
5. Clear FIFO with FCR.
6. Enable FIFO with FCR. Set FIFO trigger level if necessary.
7. Enable interrupts using IER.
8. Start data communication.

2.2 Interrupts

When interrupts occur, interrupts can be read from IIR. After resolving the interrupt, you need read the IIR again to read interrupt with next highest priority until all interrupts are resolved through ISR (Interrupt Service Routine).

2.3 Registers

RBR and THR both have the same address '0'. However, you should notice that their access permissions are different. When you read address '0', you access RBR but when you write into address '0', you access THR. Also, DLL and DLM can only be accessed when LCR[7] = 1. When LCR[7] = 0, RHR, THR and IER are accessed.

3. Programmer's Guide

The base set of registers that is used during high-speed data transfer has a straightforward access method. The extended function registers require special access bits to be decoded along with the address lines. The following guide will help with programming these registers. Note that the descriptions below are for individual register access. Some streamlining through interleaving can be obtained when programming all the registers.

Command	Action
Set Baud Rate to VALUE1, VALUE2	Read LCR, save in temp Set LCR to 80h Set DLL to VALUE1 Set DLM to VALUE2 Set LCR to temp
Set Xon1, Xoff1 to VALUE1, VALUE2	Read LCR, save in temp Set LCR to BFh Set Xon1 to VALUE1 Set Xoff1 to VALUE2 Set LCR to temp
Set Xon2, Xoff2 to VALUE1, VALUE2	Read LCR, save in temp Set LCR to BFh Set Xon2 to VALUE1 Set Xoff2 to VALUE2 Set LCR to temp
Set Software Flow Control Mode to VALUE	Read LCR, save in temp

	Set LCR to BFh Set EFR to VALUE Set LCR to temp
Set flow control threshold for 64-byte FIFO Mode	1) Set FCR to '0000_xxx1' → Set FUR to 8, set FLR to 0 2) Set FCR to '0101_xxx1' → Set FUR to 16, set FLR to 8 3) Set FCR to '1010_xxx1' → Set FUR to 56, set FLR to 16 4) Set FCR to '1111_xxx1' → Set FUR to 60, set FLR to 56
Set flow control threshold for 256-byte FIFO Mode	Set FCR to 'xxxx_xxx1' Read LCR, save in temp Set LCR to BFh Set PSR to A5h Set AFR to 01h Set FUR to Upper Threshold Value Set FLR to Lower Threshold Value Set PSR to A4h Set LCR to temp
Set TX FIFO / RX FIFO Interrupt Trigger Level for 64-byte FIFO Mode	1) Set FCR to '0000_xxx1' → Set RTR to 8, set TTR to 8 2) Set FCR to '0101_xxx1' → Set RTR to 16, set TTR to 16 3) Set FCR to '1010_xxx1' → Set RTR to 56, set TTR to 32 4) Set FCR to '1111_xxx1' → Set RTR to 60, set TTR to 56
Set TX FIFO / RX FIFO Interrupt Trigger Level for 256-byte FIFO Mode	Set FCR to 'xxxx_xxx1' Read LCR, save in temp Set LCR to BFh Set PSR to A5h Set AFR to 01h Set TTR to TX FIFO Trigger Level Value Set RTR to RX FIFO Trigger Level Value Set PSR to A4h Set LCR to temp
Read Flow Control Status	Read LCR, save in temp1 Read MCR, save in temp2

	Set LCR to ('0111_1111' **AND** temp1) Set MCR to ('0100_0000' **OR** temp2) Read FSR, save in temp3 Pass temp3 back to host Set MCR to temp2 Set LCR to temp1
Read TX FIFO / RX FIFO Count Value	Read LCR, save in temp1 Read MCR, save in temp2 Set LCR to ('0111_1111' **AND** temp1) Set MCR to ('0100_0000' **OR** temp2) Read TCR, save in temp3 Read RCR, save in temp4 Pass temp3 back to host Pass temp4 back to host Set MCR to temp2 Set LCR to temp1
Read 256-byte TX FIFO Empty Status / RX FIFO Full Status	Set FCR to 'xxxx_xxx1' Read LCR, save in temp1 Set LCR to BFh Set PSR to A5h Set AFR to 01h Set PSR to A4h Set LCR to temp1 Read ISR, save in temp2 Pass temp2 back to host
Enable Xoff Re-transmit	Read LCR, save in temp1 Set LCR to not BFh Read MCR, save in temp2 Set MCR to ('0100_0000' **OR** temp2) Set MCR to ('0100_0100' **OR** temp2) Set MCR to ('1011_1111' **AND** temp2) Set MCR to temp2 Set LCR to temp1
Disable Xoff Re-transmit	Read LCR, save in temp1 Set LCR to not BFh Read MCR, save in temp2 Set MCR to ('0100_0000' **OR** temp2) Set MCR to ('1011_1011' **AND** temp2) Set MCR to temp2

	Set LCR to temp1

Set Prescaler Value to Divide-by-1 or 4	Read LCR, save in temp1
	Set LCR to BFh
	Read EFR, save in temp2
	Set EFR to ('0001_0000' **OR** temp2)
	Set LCR to 00h
	Read MCR, save in temp3
	
	if Divide-by-1 = OK then
	Set MCR to ('0111_1111' **AND** temp3)
	else
	Set MCR to ('1000_0000' **OR** temp3)
	
	Set LCR to BFh
	Set EFR to temp2
	Set LCR to temp1

4. Sample Code

4.1 Interrupt Driven Program

The sample code illustrates a simple interrupt driven program. The program will receive input from keyboard and send the received data. The received data will be handled in ISR function when the interrupt occurs.

```
#define <stdio.h>
#define <conio.h>

#define COM1    0x03f8
#define INTVEC  0x0c

#define THR     0
#define RBR     0
#define IER     1
#define ISR     2
#define LCR     3
#define LSR     5

#define DLL     0
#define DLM     1
```

```
void interrupt (*oldportisr) ();

void interrupt ISR(void)
{
        int c;
        int data;

        //Here, we're assuming that the interrupts only occur when received data is available. To
        //handle more interrupts, we would need to read the IIR and handle them accordingly.

        //int interrupt

        //interrupt = inportb(COM1 + ISR);
        //switch(interrupt){
        //      case 0x02:
        //              Action;
        //              break;
        //      case 0x04:
        //              Action;
        //              break;
        //      ...
        //}

        do{
                //First we check LCR to see if data is available
                c = inportb(COM1 + LCR)
                if(c & 1){
                        //The data is available. So we read RBR and print it
                        data = inportb(COM1 + RBR);
                        printf("Received Data: %d", data);
                }
        } while (c & 1)

        outportb(0x20,0x20);
}

void main(void)
{
```

```
int c;

//Once we know the IRQ the next step is to find it's interrupt vector or software interrupt
//as some people may call it. Basically any 8086 processor has a set of 256 interrupt
//vectors numbered 0 to 255. Each of these vectors contains a 4 byte code which is an
//address of the Interrupt Service Routine (ISR). Fortunately C being a high level language,
//takes care of the addresses for us. All we have to know is the actual interrupt vector.
//INT   IRQ     Common Uses
// 0B     3       Serial Comms. COM2/COM4
// 0C     4       Serial Comms. COM1/COM3

//Save old interrupt vector so it can be restored in the end
oldportisr = getvect(INTVECT);
// Set up the vector using so function ISR can lead us to a set of instructions which
// would service the interrupt.
setvect(INTVEC, ISR);
//Selects which interrupts we want to disable
//The Programmable Interrupt Controller handles hardware interrupts. Most PC's will have
//two of them located at different addresses. One handles IRQ's 0 to 7 and the other
//IRQ's 8 to 15. Mainly Serial communications interrupts reside on IRQ's under 7, thus
//PIC1 is used, which is located at 0x20. Masking is done in 0x21.
//0xEF is 11101111b. The highest bit shows IRQ 7 while the lowest bit shows IRQ 0.
//Writing 0xEF would mean that we would disable all IRQs except IRQ 4.
//In case some other IRQs have been enabled already, we should read the PIC and keep
//them enabled as well. This could be done by reading the PIC and & them with 0xEF.
//Look for PIC for more information.
outportb(0x21, (inportb(0x21) & 0xEF));

//This will set baud rate.
//First we enable DLAB by setting LCR[7] = 0
//Then we set DLM and DLL to appropriate value to set desired baud rate.
outportb(COM1 + LCR, 0x80);
outportb(COM1 + DLM, 0x00);
outportb(COM1 + DLL, 0x02);

//We set LCR to set communication frame.
//8 Bits, No Parity, 1 Stop Bit.
outportb(COM1 + LCR, 0x03);
```

```
        //Reset FIFO and enable FIFO.
        outportb(COM1 + FCR, 0x07);

        /*
        //This code illustrates how to set 256-byte FIFO.
        //Enable Page 3,4
        outportb(COM1 + LCR, 0xBF); //enable page 3,4
        //Select Page 4
        outportb(COM1 + PSR, 0xA5);
        //Enable Deep FIFO
        outportb(COM1 + AFR, 0x01);
        //Set FIFO Trigger Levels
        //TX trigger level 128
        outportb(COM1 + TTR, 0x80);
        //RX trigger level 255
        outportb(COM1 + RTR, 0xFF);
        */

        //Enable Receive Interrupt
        outportb(COM1 + IER, 0x01);

        //We send the input data until ESC is pressed
        //That is it for the main function.
        //The received data will be handled by ISR function
        do {
                if (kbhit()){
                        c = getch();
                        outportb(PORT1, c);
                        }
                outportb(COM1 + THR, c);
        } while (c != 27)

        //Disable interrupt
        outportb(COM1 + IER, 0x00);
        //Mask IRQ
        outportb(0x21, (inportb(0x21) | 0x10));
        //Restore interrupt vector
        setvect(INTVECT, oldisr);
}
```

4.2 Using Global Interrupts

As there are four independent 1-channel UARTs in SB16C1054, there are four interrupts. Interrupts are assigned INT0, INT1, INT2, and INT3 for each channel. Each interrupt has six prioritized level's interrupt generation capability. The IER enables each of the six types of interrupts and INT signal in response to an interrupt generation. When an interrupt is generated, the ISR indicates that an interrupt is pending and provides the type of interrupt. SB16C1054 can handle for four interrupts with one global interrupt. Global interrupt treats four of each interrupt as one interrupt, so it is useful when external system has few interrupt resource.

Add following code to use Global Interrupt.

```
#define PSR      0
#define AFR      1
#define GICR     1
#define GISR     2
#define MCR      4

void main(void)
{
        ...
        ...
        //First we set GICR[0] = 1 to enable Global Interrupt.
        //To Access GICR, we must set LCR[7] = 0, MCR[6] = 1. This is written in the Datasheet.
        //Note that we didn't want to alter the contents of MCR other than MCR[6].
        outportb(COM1 + LCR, 0x80);
        outportb(COM1 + MCR, inportb(COM1 + MCR) | 0x40);
        //Set GICR[0] = 1 to enable Global Interrupt.
        outportb(COM1 + GICR, 0x01);
        //MCR[6] = 0
        outportb(COM1 + MCR, inportb(COM1 + MCR) & 0xBF);
        //When Global Interrupt is used, INT0 will be used as GINT. To do this, we set AFR[4] = 1.
        //To Access AFR, we set LCR = 0xBF and PSR[0] = 1.
        outportb(COM1 + LCR, 0xBF);
        outportb(COM1 + PSR, 0xA5);
        //We set AFR[4] = 1.
        //Also, we set Global Interrupt Polarity with AFR[5] = 1. This means, GINT pin outputs '1'
```

```
        //when interrupt is generated
        outportb(COM1 + AFR, 0x30);
        …
        …
}

void interrupt ISR(void)
{
        //When interrupt is generated through GINT, we should find out which of the four UARTs
        //generated the interrupt. To do this, we read GISR.
        int InterruptNumber;

        //Reading GISR requires same procedure as reading writing GICR.
        outportb(COM1 + LCR, 0x80);
        outportb(COM1 + MCR, inportb(COM1 + MCR) | 0x40);
        //Read GISR
        InterruptNumber = inportb(COM1 + GISR);
        outportb(COM1 + MCR, inportb(COM1 + MCR) & 0xBF);

        //Depending on which UART generated the interrupt, we would implement different logic
        if(InterruptNumber & 0x01){
        }
        else if(InterruptNumber & 0x02){
        }
        else if(InterruptNumber & 0x04){
        }
        else if(InterruptNumber & 0x08){
        }
        …
        …
}
```

AN16003. The Benefits of UART with Deep FIFO

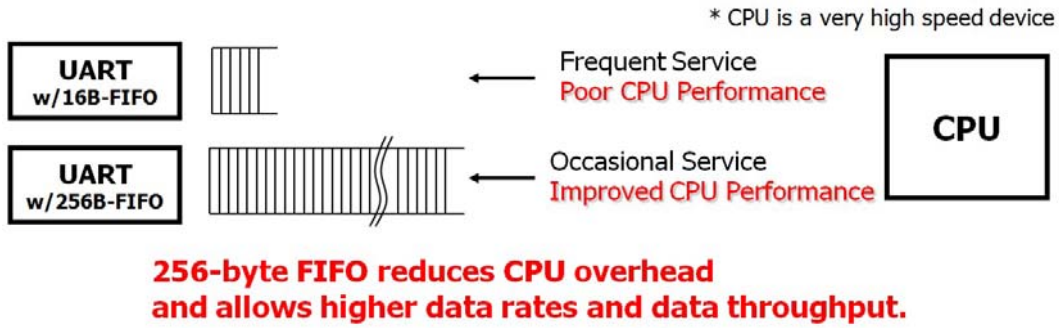
(for SystemBase's the Enhanced UART Core, SB16C105x)

1. Overview

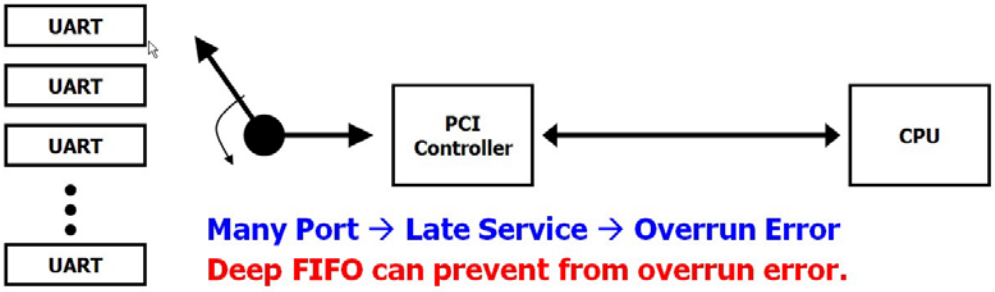
Since UART offer more functions after their coming out, FIFO sizes of UART are also increased more and more. In this application note, we are going to explain how the size of FIFO influences to systems. Our company made a 16 ports serial card composed of Octal-UART with 256-Byte FIFO, SB16C1058 which is developed recently; we'll report how the function of serial card are improved

2. Deep FIFO

UART is a communication controller which operates slowly compared to CPU. So UART has FIFO memory in their internal circuit to reduce service time of CPU. If there is no internal FIFO, CPU must service UART, whenever UART transfers/ receives one byte to/from CPU. This phenomenon indicates that CPU can't do other work reducing performance of system. To improve performance of serial communication system, FIFO sizes are broadening more and more.



When developing serial communication systems, the case to need many serial channels would be happens. Serial Multiport and Device server are the good example.



As the above diagram, the UART channels are existed, the more performances of CPU are needed on account of servicing to UART channels. So the performance of system is decreased when a CPU have many serial channels to deal with. Moreover CPU may need much time to service to UART channels when the baud rates get fasted. Because interrupts happens frequently when the baud rate get fasted.

If CPU has to service to many UART channels, some UART channels might have FIFO of full data during waiting CPU services. This crisis situation causes Overrun Error which is induced by Overwriting to full FIFO. However, if UART has broad size FIFO, Overrun Errors are prevented since there are many space to store data during servicing of channels. This is crucially important in multi-channel communication system.

3. The Statistics of Deep FIFO Effect

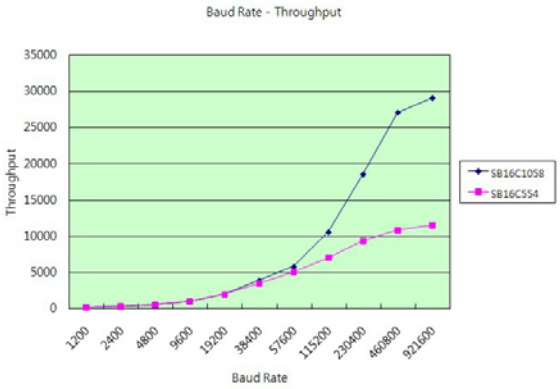
To test above descriptions, we made two types of 16 ports Serial Card. One adopted SB16C554A which has 16-byte FIFO. Another adopted SB16C1058 which has 256-byte FIFO. We compare the throughput and CPU usage rate between two multi channel serial cards.

Multi-16C/PCI using SB16C554A
with 16-byte FIFO

Multi-16C/PCI using SB16C1058
with 256-byte FIFO

3.1 Comparative table of the throughputs between 16B and 256B FIFO.

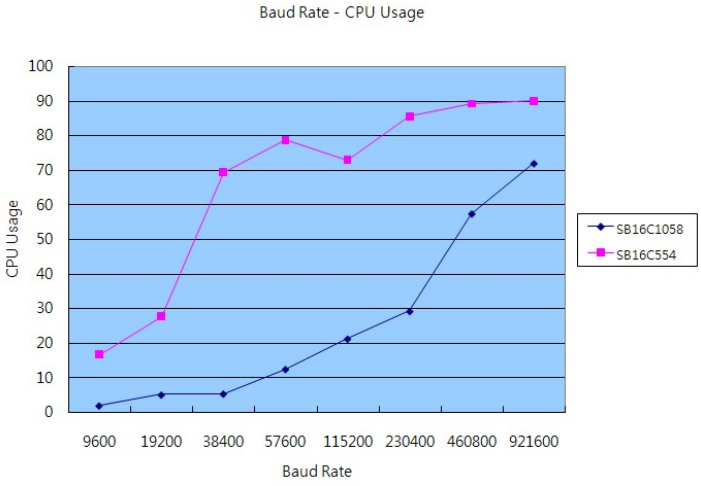
Baud Rate	SB16C1058 (256-byte FIFO)	SB16C554 (16-byte FIFO)
1200 bps	120 cps	120 cps
2400 bps	240 cps	240 cps
4800 bps	480 cps	480 cps
9600 bps	960 cps	960 cps
19200 bps	1920 cps	1905 cps
38400 bps	3840 cps	3400 cps
57600 bps	5740 cps	5000 cps
115200 bps	10500 cps	7000 cps
230400 bps	18500 cps	9300 cps
460800 bps	27000 cps	10800 cps
921600 bps	29000 cps	11500 cps



As we can see above data, we can find there is great difference of the throughput in high baud rate. When the baud rate is increased, the multi-channel serial card have Overrun Errors. However, Overrun Error is decreased greatly if the multi-channel Serial Card use UART with deep FIFO included broad FIFO size.

3.2 The comparative table of the CPU usage between 16-byte FIFO and 256-byte FIFO.

Baud Rate	SB16C1058 (%)	SB16C554 (%)
9600 bps	1.802	16.494
19200 bps	4.935	27.631
38400 bps	5.23	69.316
57600 bps	12.341	78.707
115200 bps	21.126	72.888
230400 bps	29.129	85.586
460800 bps	57.342	89.289
921600 bps	71.892	90.052



As we can see upper data, when the baud rate is increased, the CPU usage can become bigger. But the enhanced UART with deep FIFO can make more efficient systems.

4. Conclusions

Now SystemBase offers the best scalability in serial communication industry and have UART with the largest FIFO.

If you want to develop serial multi-channel communication system with high performance, it is best solution to select Systembase semiconductor.

SystemBase's UART with Deep FIFO provides bellow features.

- CPU Usage is decreased so that system performances get raised.
- The throughput of serial communication is better.
- Overrun Error can be decreased.

AN16004. Interfacing between UART and CPU

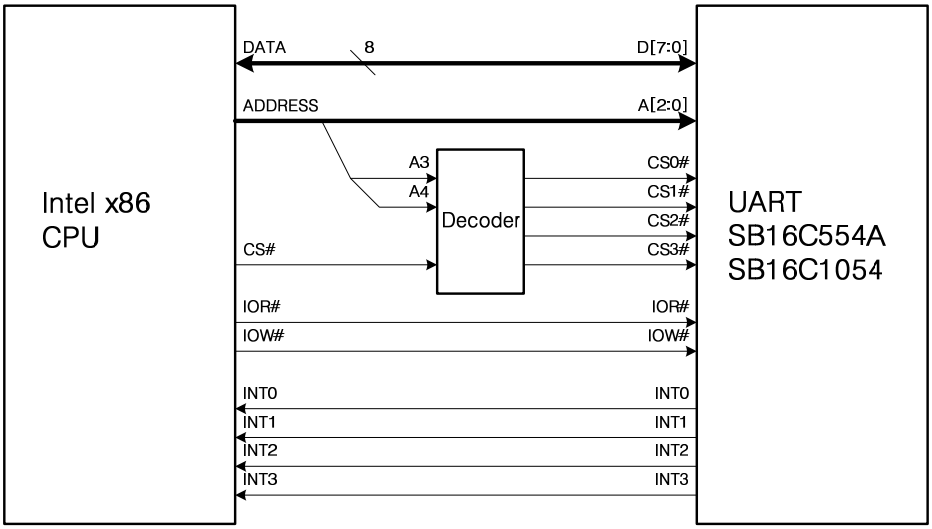
(for SystemBase's UART, SB16C554A and SB16C1054)

1. Overview

This document explains methods which enable of user to connect CPU and UART of SystemBase.

This content deals with bus signals connecting techniques on general Intel CPU system bus with SB16C554A or SB16C1054.

2. Bus Connection



① Data Bus

Generally, UART use 8-bit data bit so that it is proper to connect only 1-Byte.

② Address Bus & Chip Selection

Since UART has eight register spaces in each channel, it is possible to access using three address lines.

But, since Quad-UART, SB16C554A and SB16C1054 have 4 channels in one-chip, you need 2 more address to assign each channel. Accordingly, you can access 4 channels UART using 5-bits address lines.

Low 3-bit of Address, A[2:0], are used for assigning internal registers of one UART channel and high 2-bit, A[4:3], are used for selecting one UART channel among four UART. To implement these selection, we have to use decoder circuits. Bellow is the VHDL code relating decoder.

```
CS0# <= '0' when CS# ='0' and EA(4 downto 3)="00" else '1'
CS1# <= '0' when CS# ='0' and EA(4 downto 3)="01" else '1'
CS2# <= '0' when CS# ='0' and EA(4 downto 3)="10" else '1'
CS3# <= '0' when CS# ='0' and EA(4 downto 3)="11" else '1'
```

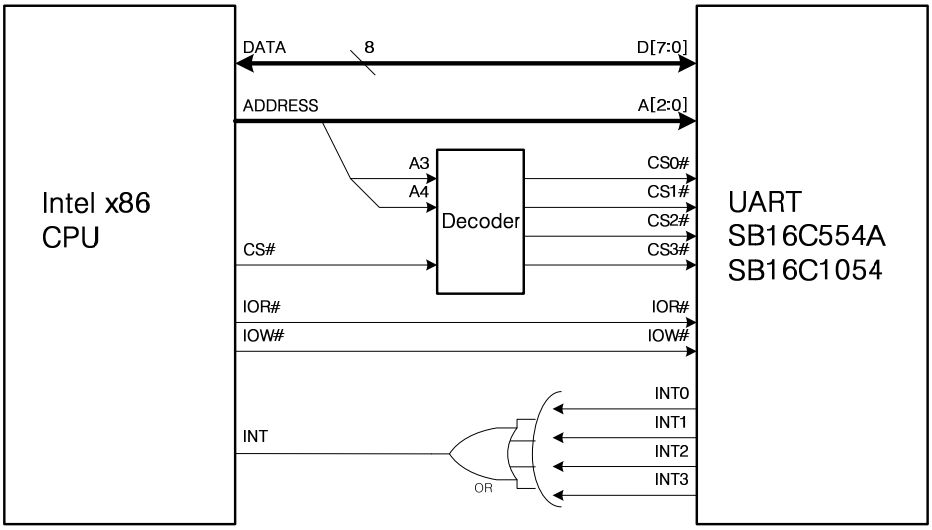
When you have to use Octal-UART with one chip select signal, you can also use above address decoding method.

③ IOR# & IOW#

Early x86 CPUs have each Read Strobe and Write Strobe for I/O and Memory. But, since UART is a I/O device, it utilizes I/O Read strobe and I/O write strobe.

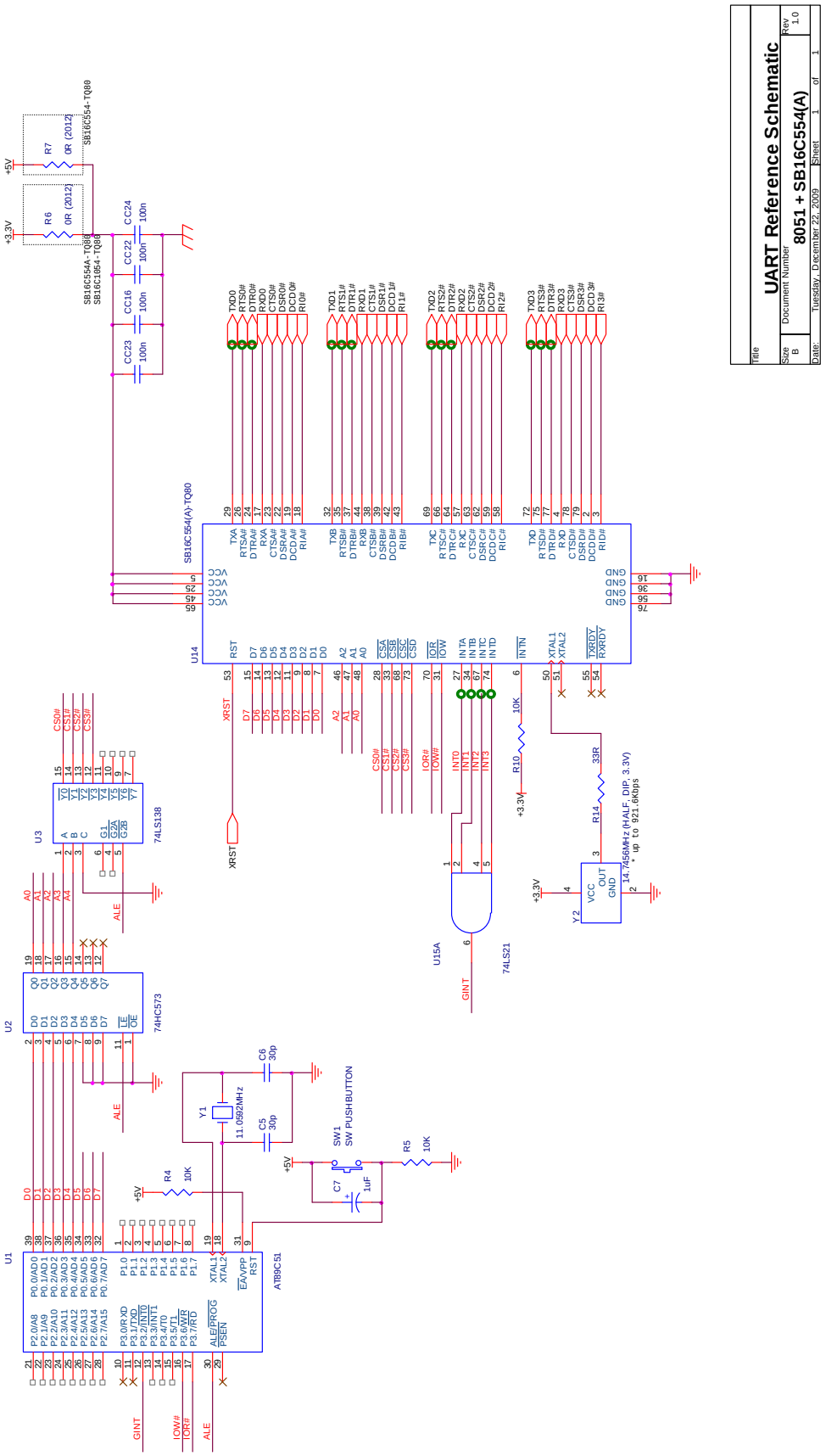
④ Interrupt

There are 4 Interrupt request signals because our Quad-UART has 4 channels. If there are redundant INT pins, you can utilize it by 1:1 connection like above figure. This connection enable direct search selected UART channel because it can find directly what channel generate interrupts.
But if there are limited interrupt pins, you can make logic OR using 4 interrupt inputs, INT0~INT3, to get one global interrupt output.



But when a global interrupt occurs, it is unknown which interrupt of the four UART channels occurred from. So interrupts should be handled after checking each UART channel.

3. Reference Schematic



SystemBase