Serial Communication Interface (SCI)

To exchange data between the MCU and other external peripherals such as the 7-segment display or lcd display, we use parallel data transfer. However, parallel data transfer requires many I/O pins. Also, the longer the distance, the more difficult the parallel data transfer becomes in terms of synchronization, because the parallel bits need to arrive around the same time on the other end.

The SCI is one of the existing means of serial communication in the micro board. We will later study other synchronous serial communication protocols such as SPI and I2C. One unique feature of the SCI is that it allows us to transfer data asynchronously using only 2 pins, aside from the signal ground: Rx and Tx, not needing a clock pin. In order for this to work, both ends need to use the same Baud Rate (speed).

The SCI was designed to operate utilizing the industrial standard TIA-232, most commonly known as RS-232. The TIA-232 standard most updated revision (revision F) was published in 1997 by the Telecommunications Industry Association.

While the microcontroller is 5[V] logic (0[V] for logic 0 and 5[V] for logic 1), the TIA-232 standard defines different voltage levels, therefore we need an RS232 transceiver to interface the sci with the outside world.

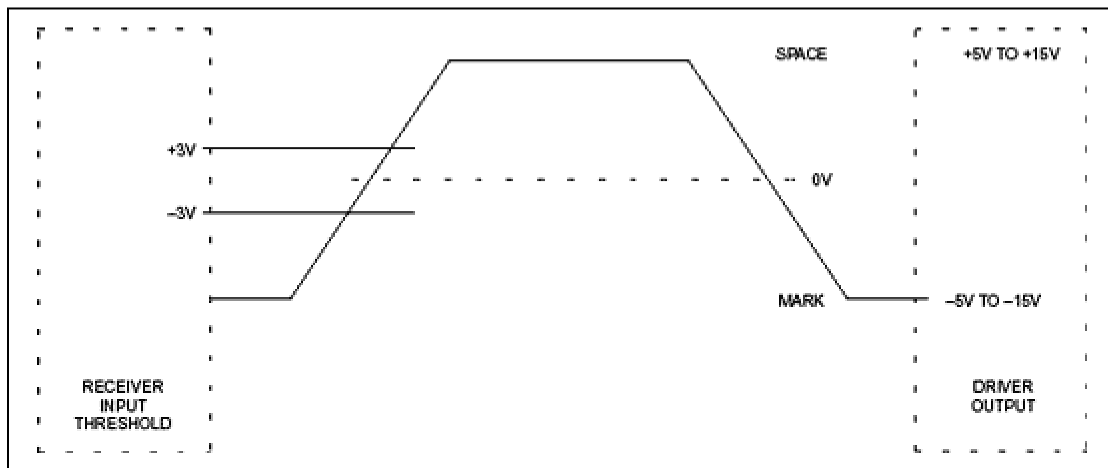


Figure 1- TIA-232 Voltage Levels

The RS232 transceiver can be identified in the board schematics, it’s labeled as IC2. The RS232 transceiver outputs then connect to the DB9 port on the back of the board. From the DB9 port, we can connect to a computer using a regular RS232 cable. Most recent computers most likely will not have a native RS232 port, in which case an RS232 to USB adapter will be required.

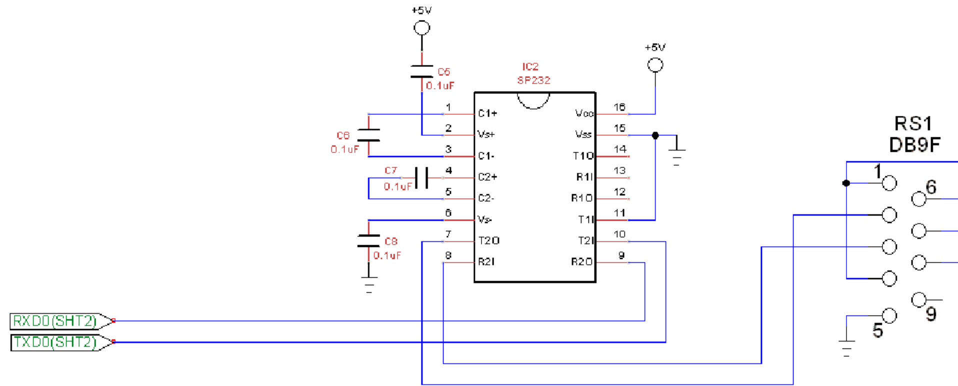


Figure 2- RS232 Transceiver

Figure 3- RS232 Cable

Figure 4- USB to RS232 Adapter

Configuring the SCI

The microcontroller has more than one SCI. According to the schematics, only SCI0 is connected to an RS232 transceiver and therefore to the DB9 port that we will connect to the computer. We can also see in the schematics that the SCI1 is connected to an Infrared transceiver (IC1) that you can find beside the DB9 connector (to the left). We will not have time to cover the infrared transceiver during this course, but you could experiment with it in the future.

There are Five more SCIs: SCI1, SCI2, SC3, SCI4 and SCI5. If we look at the peripherals diagram in page 35 of the datasheet or in the course notes diagram, we could find them. Some of them are multiplexed with other peripherals, so to be able to use them may require extra register configuration. To locate the pins of the one we are going to work with, we look for PS0 (RXD0) and PS1 (TXD0).

- PS0 (RXD0) is PIN 89
- PS1 (TXD0) is PIN 90

Figure 1-1. MC9S12XD-Family Block Diagram

Figure 5- SCI0- SCI5 Location

Configuring the SCI0

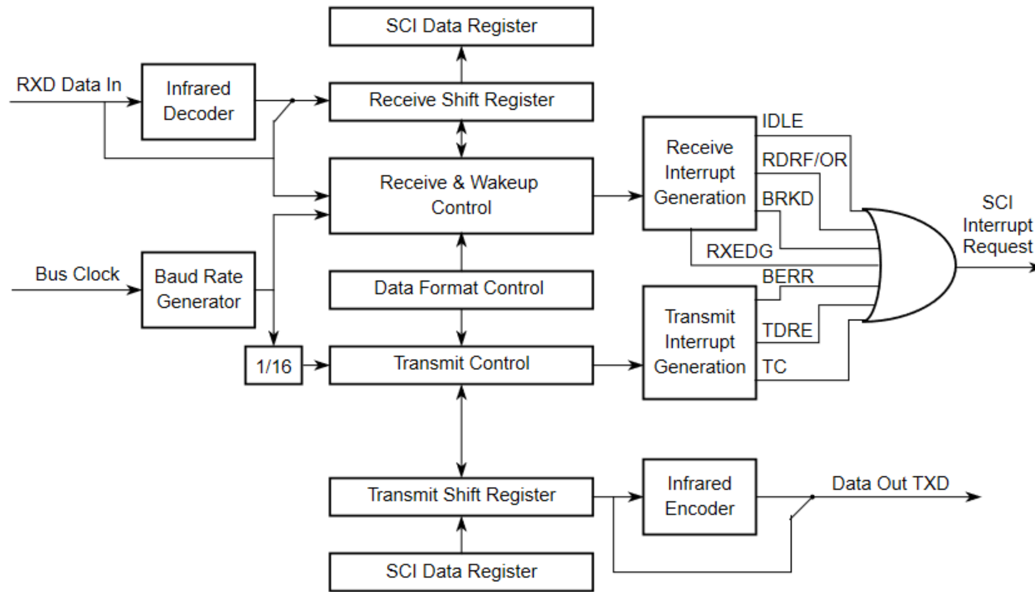
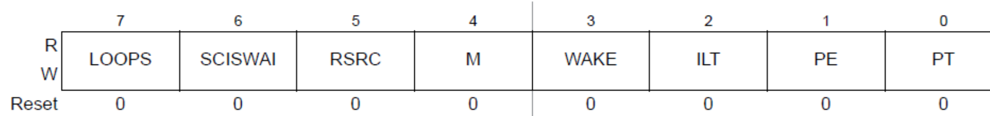


Figure 6 - SCI Block Diagram

There are mainly 3 registers that we will need to configure to Initialize the SCI0 peripheral, although the first register we will just leave it defaulted to zero. Please note the registers we are working with are named SCI0xxx. The datasheet displays the general names for the registers, noting that they would work the same way for any SCI peripheral: SCI0xxx, SCI1xxx, SCI2xxx…etc.

Control Register 1: SCI0CR1

11.3.2.2 SCI Control Register 1 (SCICR1)



	7	6	5	4	3	2	1	0
R W	LOOPS	SCISWAI	RSRC	M	WAKE	ILT	PE	PT
Reset	0	0	0	0	0	0	0	0

Figure 11-5. SCI Control Register 1 (SCICR1)

- All the default settings work here.

Table 11-3. SCICR1 Field Descriptions

Field	Description
7 LOOPS	**Loop Select Bit** — LOOPS enables loop operation. In loop operation, the RXD pin is disconnected from the SCI and the transmitter output is internally connected to the receiver input. Both the transmitter and the receiver must be enabled to use the loop function. 0 Normal operation enabled 1 Loop operation enabled The receiver input is determined by the RSRC bit.
6 SCISWAI	**SCI Stop in Wait Mode Bit** — SCISWAI disables the SCI in wait mode. 0 SCI enabled in wait mode 1 SCI disabled in wait mode
5 RSRC	**Receiver Source Bit** — When LOOPS = 1, the RSRC bit determines the source for the receiver shift register input. See Table 11-4. 0 Receiver input internally connected to transmitter output 1 Receiver input connected externally to transmitter
4 M	**Data Format Mode Bit** — MODE determines whether data characters are eight or nine bits long. 0 One start bit, eight data bits, one stop bit 1 One start bit, nine data bits, one stop bit
3 WAKE	**Wakeup Condition Bit** — WAKE determines which condition wakes up the SCI: a logic 1 (address mark) in the most significant bit position of a received data character or an idle condition on the RXD pin. 0 Idle line wakeup 1 Address mark wakeup

Control Register 2: SCI0CR2

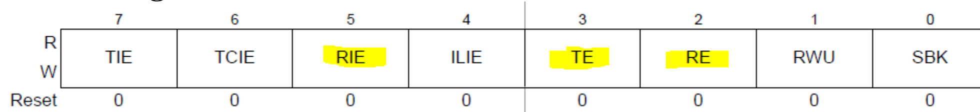


Figure 11-9. SCI Control Register 2 (SCICR2)

- We need to enable receiver and transmitter: Set (1) BIT2 and BIT3
- If we like to use interrupts for the receiver: Set (1) BIT5

Table 11-9. SCICR2 Field Descriptions

Field	Description
7 TIE	**Transmitter Interrupt Enable Bit** — TIE enables the transmit data register empty flag, TDRE, to generate interrupt requests. 0 TDRE interrupt requests disabled 1 TDRE interrupt requests enabled
6 TCIE	**Transmission Complete Interrupt Enable Bit** — TCIE enables the transmission complete flag, TC, to generate interrupt requests. 0 TC interrupt requests disabled 1 TC interrupt requests enabled
5 RIE	**Receiver Full Interrupt Enable Bit** — RIE enables the receive data register full flag, RDRF, or the overrun flag, OR, to generate interrupt requests. 0 RDRF and OR interrupt requests disabled 1 RDRF and OR interrupt requests enabled
4 ILIE	**Idle Line Interrupt Enable Bit** — ILIE enables the idle line flag, IDLE, to generate interrupt requests. 0 IDLE interrupt requests disabled 1 IDLE interrupt requests enabled
3 TE	**Transmitter Enable Bit** — TE enables the SCI transmitter and configures the TXD pin as being controlled by the SCI. The TE bit can be used to queue an idle preamble. 0 Transmitter disabled 1 Transmitter enabled
2 RE	**Receiver Enable Bit** — RE enables the SCI receiver. 0 Receiver disabled 1 Receiver enabled
1 RWU	**Receiver Wakeup Bit** — Standby state 0 Normal operation. 1 RWU enables the wakeup function and inhibits further receiver interrupt requests. Normally, hardware wakes the receiver by automatically clearing RWU.
0 SBK	**Send Break Bit** — Toggling SBK sends one break character (10 or 11 logic 0s, respectively 13 or 14 logics 0s if BRK13 is set). Toggling implies clearing the SBK bit before the break character has finished transmitting. As long as SBK is set, the transmitter continues to send complete break characters (10 or 11 bits, respectively 13 or 14 bits). 0 No break characters 1 Transmit break characters

Baud Rate Register: SCI0BD

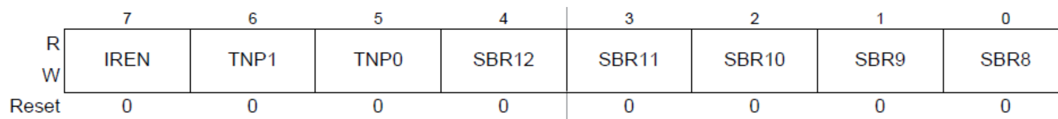


Figure 11-3. SCI Baud Rate Register (SCIBDH)

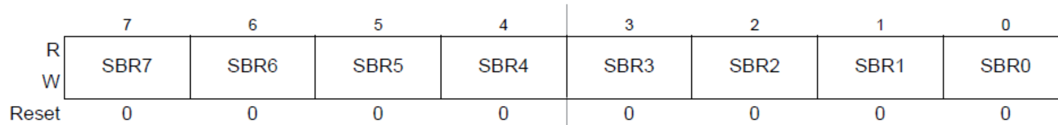


Figure 11-4. SCI Baud Rate Register (SCIBDL)

Although this register is separated into two 8-bit registers, we will use the 16-bit access directly through the **SCI0BD** register.

- We leave BIT7 – BIT5 set as default 0
- We enter the baud rate setting in: BIT12 – BIT0

Table 11-1. SCIBDH and SCIBDL Field Descriptions

Field	Description
7 IREN	**Infrared Enable Bit** — This bit enables/disables the infrared modulation/demodulation submodule. 0 IR disabled 1 IR enabled
6:5 TNP[1:0]	**Transmitter Narrow Pulse Bits** — These bits enable whether the SCI transmits a 1/16, 3/16, 1/32 or 1/4 narrow pulse. See Table 11-2.
4:0 7:0 SBR[12:0]	**SCI Baud Rate Bits** — The baud rate for the SCI is determined by the bits in this register. The baud rate is calculated two different ways depending on the state of the IREN bit. The formulas for calculating the baud rate are: When IREN = 0 then, SCI baud rate = SCI bus clock / (16 x SBR[12:0]) When IREN = 1 then, SCI baud rate = SCI bus clock / (32 x SBR[12:1]) **Note:** The baud rate generator is disabled after reset and not started until the TE bit or the RE bit is set for the first time. The baud rate generator is disabled when (SBR[12:0] = 0 and IREN = 0) or (SBR[12:1] = 0 and IREN = 1). **Note:** Writing to SCIBDH has no effect without writing to SCIBDL, because writing to SCIBDH puts the data in a temporary location until SCIBDL is written to.

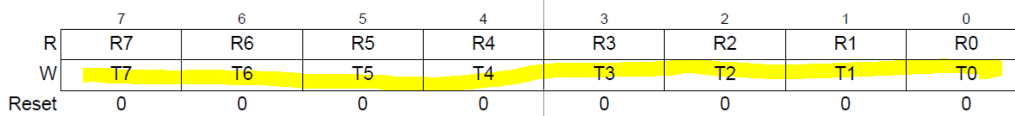
Since we will not be using the Infra-Red (IREN = 0), our baud rate setting will be as:

$$\boldsymbol{SBR[12:0] = \frac{BUS\ Speed}{16 \times Desired\ BR}}$$

We approximate the result to the nearest integer value.

Writing to the SCI0 (Transmitting)

To write data to the SCI, we need to write the **SCI0DRL** register.



	7	6	5	4	3	2	1	0
R	R7	R6	R5	R4	R3	R2	R1	R0
W	T7	T6	T5	T4	T3	T2	T1	T0
Reset	0	0	0	0	0	0	0	0

Figure 11-13. SCI Data Registers (SCIDRL)

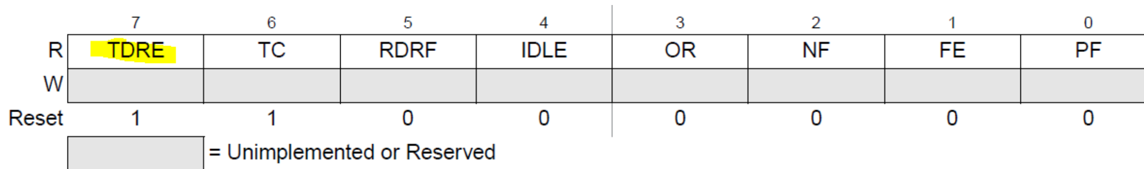
Read: Anytime; reading accesses SCI receive data register

Write: Anytime; writing accesses SCI transmit data register; writing to R8 has no effect

Table 11-12. SCIDRH and SCIDRL Field Descriptions

Field	Description
SCIDRH 7 R8	**Received Bit 8 —** R8 is the ninth data bit received when the SCI is configured for 9-bit data format (M = 1).
SCIDRH 6 T8	**Transmit Bit 8 —** T8 is the ninth data bit transmitted when the SCI is configured for 9-bit data format (M = 1).
SCIDRL 7:0 R[7:0] T[7:0]	**R7:R0 —** Received bits seven through zero for 9-bit or 8-bit data formats **T7:T0 —** Transmit bits seven through zero for 9-bit or 8-bit formats

Before we write data to the register, we need to make sure the register is ready to receive data. To do that, we check the **TDRE** bit of the **SCI0SR1** register. We wait while the flag is ‘0’.



	7	6	5	4	3	2	1	0
R	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF
W								
Reset	1	1	0	0	0	0	0	0

= Unimplemented or Reserved

Figure 11-10. SCI Status Register 1 (SCISR1)

7 TDRE	**Transmit Data Register Empty Flag —** TDRE is set when the transmit shift register receives a byte from the SCI data register. When TDRE is 1, the transmit data register (SCIDRH/L) is empty and can receive a new value to transmit.Clear TDRE by reading SCI status register 1 (SCISR1), with TDRE set and then writing to SCI data register low (SCIDRL). 0 No byte transferred to transmit shift register 1 Byte transferred to transmit shift register; transmit data register empty

Example

```
//Blocking 1 byte transmission
while(!(SCI0SR1 & SCI0SR1_TDRE_MASK));//Wait till transmit data register is empty
SCI0DRL = data;

//Non-locking 1 byte transmission
if(SCI0SR1 & SCI0SR1_TDRE_MASK) //Check if transmit data register is empty
{
      SCI0DRL = data;
}
```

Reading from the SCI0 (Receiving)

To read data from the SCI, we need to read the **SCI0DRL** register. Please note that the receiving and transmitting happens in the same register. How is this possible? Maybe looking at *figure 6* can give us a clue.

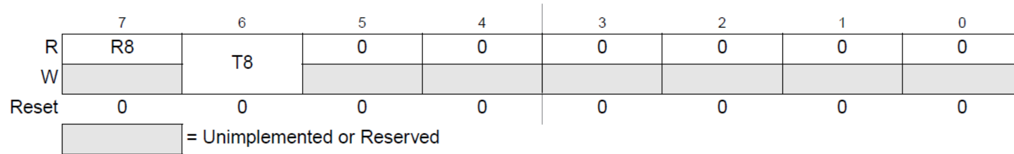


Figure 11-12. SCI Data Registers (SCIDRH)

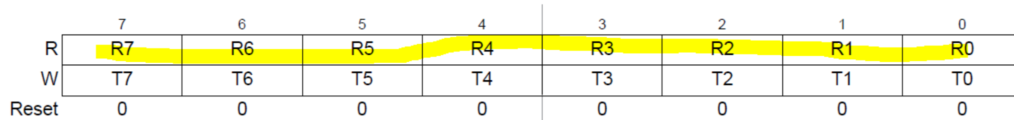


Figure 11-13. SCI Data Registers (SCIDRL)

Table 11-12. SCIDRH and SCIDRL Field Descriptions

Field	Description
SCIDRH 7 R8	**Received Bit 8** — R8 is the ninth data bit received when the SCI is configured for 9-bit data format (M = 1).
SCIDRH 6 T8	**Transmit Bit 8** — T8 is the ninth data bit transmitted when the SCI is configured for 9-bit data format (M = 1).
SCIDRL 7:0 R[7:0] T[7:0]	**R7:R0** — Received bits seven through zero for 9-bit or 8-bit data formats **T7:T0** — Transmit bits seven through zero for 9-bit or 8-bit formats

Before we can read the data from the register, we need to make sure there is data ready to be read. To do that, we check the **RDRF** flag in the SCI Status Register 1, **SCISR1**.

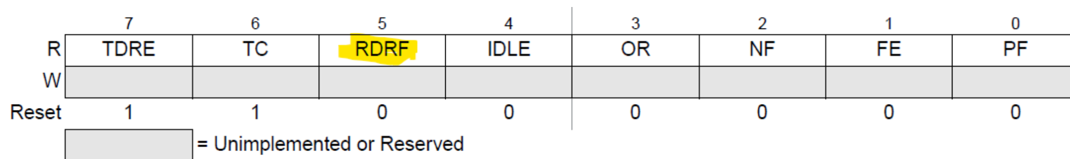


Figure 11-10. SCI Status Register 1 (SCISR1)

5 RDRF	**Receive Data Register Full Flag** — RDRF is set when the data in the receive shift register transfers to the SCI data register. Clear RDRF by reading SCI status register 1 (SCISR1) with RDRF set and then reading SCI data register low (SCIDRL). 0 Data not available in SCI data register 1 Received data available in SCI data register

There are two options for knowing if there is data available in the register:

- Poll the flag, then as the register information describes, checking such flag and then reading the register clears the flag automatically.

  Example

```
unsigned char data;

//Blocking receiving
while(!(SCI0SR1 & SCI0SR1_RDRF_MASK)); //Wait till a character is received
data = SCI0DRL;

//Non-blocking receiving
if(SCI0SR1 & SCI0SR1_RDRF_MASK) //Check if a character has been received
{
	data = SCI0DRL;
}
```

- Enabling interrupt by setting the **RIE** bit in the **SCI0CR2** register. Once a character is received, the flag being active will trigger the interrupt, then in the ISR (Interrupt Service Routine). To read the data and clear the flag, we mask the **RDRF** bit of the **SCI0SR1** register, then we need to read the data from the **SCI0DRL** register.

  Example

```
interrupt VectorNumber_Vsci0 void Vsci0_ISR(void)
{
  if(SCI0SR1 & SCI0SR1_RDRF_MASK)
  {
    data = SCI0DRL;
  }
}
```

Pointers

A pointer is a variable that holds the memory address of another variable, constant or literal. They allow us to perform a call by reference in a way that variables outside the scope of the function can be modified within such function. They are also useful for managing strings as an array of characters, where the pointer can "point" to the beginning of such array or any specific character within the array.

Declaring a Pointer Type

Declaration:

*type * identifier;* Where *type* can be any atomic (int, char, etc) or a custom type previously defined using a typedef, and *identifier* is the name of the pointer variable.

We say that the actual type of the pointer is not *type,* but *type **. For instance:

Int pMyInt* is a pointer to a variable of type *int,* therefore the type of such pointer is *int** and holds the address of an *int* variable.

Assigning a pointer

Once we have declared a pointer, we can then assign an address of a variable or constant / literal to it. To do that, we use the address *(&)* operator. For instance:

Int MyInt;

Int pMyInt = &MyInt;*

Obtaining the Content of the Variable Pointed

Once we have declared and assigned a pointer, we can now reference the variable using the pointer name with asterisk in front of it, in which case we can read such variable or assign a new value to it. For instance:

Int MyInt;

Int pMyInt = &MyInt;*

**pMyInt = 5; // Assigns 5 to the variable being pointed by pMyInt, MyInt*

*Int sum = *pMyInt + 2 //Assigns 5+2 into sum*

Using Pointers to Structures

We can access the structure through the pointer, just like any variable, then we can reference the structure member using the accessor: "." Operator. For instance:

```
typedef struct MyStruct_
{
    int Member1;
    float Member2;
} MyStruct;
MyStruct record;
MyStruct * pMyStruct = &record;
*pMyStruct.Member1 = 5;
*pMyStruct.Member2 = 2.5;
```

Since accessing structure elements via pointers is common and useful, an alternative and easier to understand syntax is available:

```
pMyStruct->Member1 = 5;
pMyStruct->Member2 = 2.5;
```

Operations with pointers

Since a pointer variable holds an address, it therefore holds a number, which could be incremented or decremented, meaning that incrementing such pointer would make it point to the next address, or decrementing it, would make it point to the previous address; the addresses would be incrementing or decrementing a number of bytes equal to the size of the variable that is being pointed. For instance, a pointer to a *char* variable would increment /decrement by 1 byte, while a pointer to an *int* (16 bytes) would make increments / decrements of 2 bytes.

This type of operation can be extremely dangerous as we could make the pointer point to a wrong address by mistake, so caution must be taken with using pointer arithmetic.