Interfacing of peripherals

Interfacing chips are needed to solve:
1-the speed problem
2-synchrouientis for data transfer between CPU & I/O device
* The primary function of the µp is to accept data from input, read instructions from memory, process data according to the instructions and send result to output using logic circuit. (Hardware) and writing instructions (software) to enable the µp to comm. With these peripherals is called interacting.
The µp communication with peripherals in either of two formats: asynchronous or synchronous. Data transfer between µp & its peripherals can take place under various conditions as shown below**:**
An interrupt is an external or internal event that interrupts the µp to inform it that a device needs its service.

Format of data transfer (synch & asynch)

- Synchronous means at the same time, the TX & RX are synchrouitel with same clock
- Asynchronous means at irregular interval, synch used for high speed data transfer while asynch. Used for low-speed data transfer

Modes of data transfer

µp receives or transmits binary data either parallel or serial, in parallel mode, the entire word (4bit, 8bit, 16bit) is transferred at one. In serial mode data transferred one bit at a time over a single line between µp & peripheral. A word is converted into stream of eight bits this called to parallel conversion, then a stream of 8 bit is converted to parallel to serial conversion.

Condition of data transfer

The process of data transfer between µp & peripheral is controlled either by µp or by peripheral.

µp controlled data transfer

Most peripheral respond slowly in comparison with the speed of the µp, it can take five different conditions:
1-**Unconditional data transfer:** in this form of data transfer, the µp assumes that the peripherals are always available, ex: data, and goes on to execute the next instruction.
2-**data transfer with polling:** (status check): the µp is kept in a loop to check whether data are available, this is called polling, ex: to read from input keyboard, the µp can keep polling the port until is pressed.
3-data transfer with interrupts: in this condition where the peripheral is ready to transfer data, it's sends an interrupt signal to the µp. The µp stops the executions of the program accept the data from the peripheral, and then returns to the program.
4-data transfer with ready signal: when peripheral response time is slower than µp time, the ready signal can be used to add T-states, thus extending the execution time this processor provides sufficient time for the peripheral to complete the data transfer.

5-data transfer with handshake signals: in this data transfer signals are exchanged between the µp & peripheral prior to actual data transfer, these signals are called handshake signals.
The function of these signals is to:
1-ensure the readiness of peripheral.
2-to synchronize the timing of data transfer for example:
A / D is used as input device, the µp need to wait because of the slow conversion, at the end of conversion the A / D converter send a data ready (DR) also known as end acknowledge by sending a signal to the converter.

<u>Peripheral-controlled data transfer</u>
This is device-controlled I / O this type of data transfer is employed when the peripheral is much faster than the µp.
For example in case of direct memory access (DMA) controller sends a hold signal to µp, the µp releases its data bus and the address bus to the DMA controller, and the data are transferred at high speed.

I/O Operations

There are three basic ways to get data into or out of a memory. They are called programmed I/O, interrupt-driven I/O, and direct memory access (DMA). Although programmed I/O is the lowest of the three, it is used in simpler microprocessor systems where speed is unimportant. As the system becomes more complex, the interrupt approach becomes necessary. In the most advanced systems, DMA is needed because it is the only way to transfer large amounts of data in a short time.

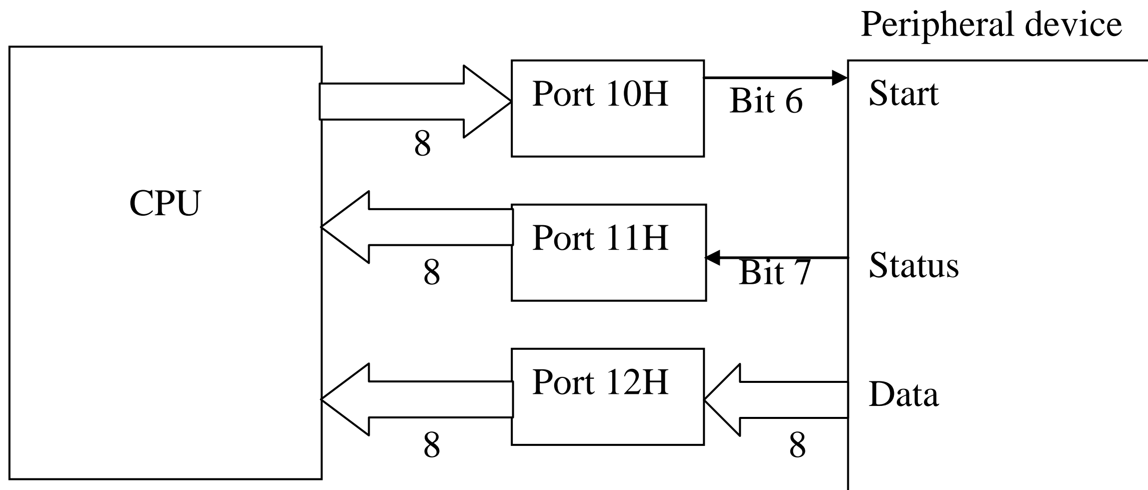
Programmed I/O

Programmed I/O uses instructions to get data into or out of a CPU. To correctly time the data transfers, programmed I/O relies either on clock timing (synchronous) or on handshaking (asynchronous).
IN 8-bit port address: The contents of the input port designated in the operand (8-bit port address) are read and loaded into the accumulator.
Out 8-bit port address: The contents of the accumulator are copied into the I/O port specified by the operand.

No.	Instruction	Type	No. of Bytes	Function	Effect
1.	IN 8-bit address	I/O	2	A=byte Ex. In 55; A=byte	None
2.	Out 8-bit address	I/O	2	Port address ← A Ex. Out 53; 53 ← A	None

Programmed Input

The following figure shows a CPU connected to ports 10H, 11H, and 12H. Bit 6 of port 10H is connected to the start input of the peripheral device, and bit 7 of port 11H to the status output. The device can send data to the CPU through input port 12H.



The basic idea is this. When the CPU is ready to input a word, it sends a high start bit to the peripheral device. When the device has the data ready for transfer, it sends a high status bit to port 11H. Until the status bit is high, the CPU waits. As soon as the status bit goes high, the CPU inputs one byte of data.

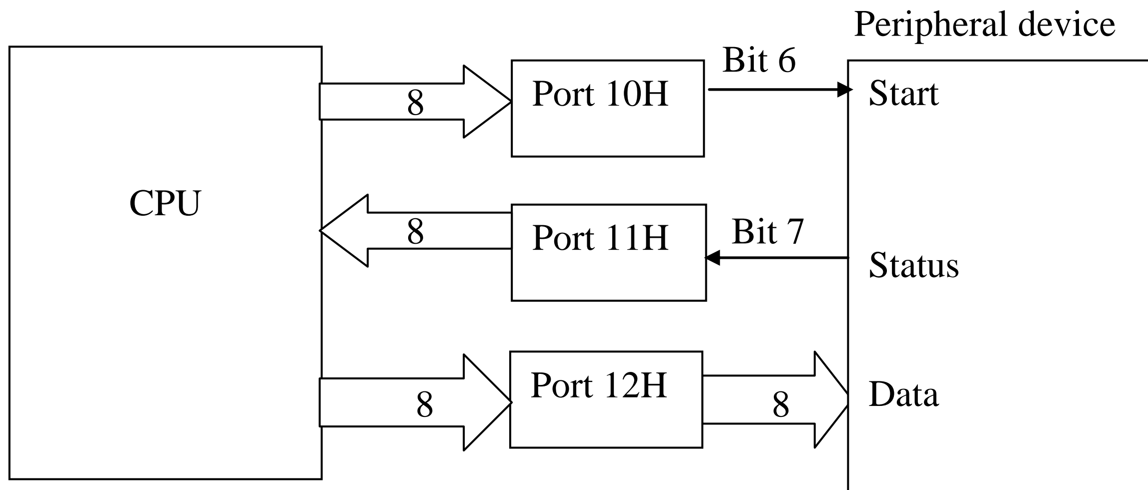
Suppose we want to input 256 bytes and store them at address 4000H through 40FFH. Here is an example of programmed I/O written for an 8085 system:

Label	Mnemonic	Comment
	LXI H, 4000H	; initialize HL pointer
	MVI C, 00H	; initialize counter
LOOP:	MVI A, 40H	; set start bit
	OUT 10H	; send high start bit
WAIT:	IN 11H	; get status bit
	ANI 80H	; isolate status bit
	JZ WAIT	; wait if device not ready
	IN 12H	; input data
	MOV M, A	; store data
	INX H	; update HL pointer
	MVI A, 00H	; reset start bit
	OUT 10H	; send low start bit
	DCR C	; count down
	JNZ LOOP	; go back if not finished
	HLT	

Notice in the previous figure that only one bit of port 10h is used; the other 7 bits are don't cares. Likewise in port 11h, only 1 bit is used. Ports 10h and 11h are necessary for the handshaking operations; port 12h is needed for the data transfer.

Programmed Output

The following figure shows a CPU connected to handshaking ports 10h and 11h. It is also connected to output port 12h. As before, bit 6 of port 10h is the start bit, and bit 7 of port 11h is the status bit.



Here is the procedure for output operations. When the CPU is ready, it will latch the data into port 12h. Then, the CPU sends a high start bit to indicate that valid data is waiting for transfer. After the peripheral device has loaded the data, it returns a high status bit.

Suppose we want to output 256 bytes from memory locations 4000h to 40ffh. Here is an example of programmed output:

Label	Mnemonic	Comment
	LXI H, 4000H	; initialize HL pointer
	MVI C, 00H	; initialize counter
LOOP:	MOV A, M	; get next byte
	OUT 12H	; latch data into port 12h
	MVI A, 40H	; set start bit
	OUT 10H	; send high start bit
WAIT:	IN 11H	; get status bit
	ANI 80H	; isolate status bit
	JZ WAIT	; wait if device not ready
	INX H	; update HL pointer
	MVI A, 00H	; reset start bit
	OUT 10H	; send low start bit
	DCR C	; count down
	JNZ LOOP	; go back if not finished
	HLT	

Incidentally, programmed I/O is sometimes referred to as polled I/O. In the examples given, we have used software to control the I/O transfers of a single peripheral device. By modifying the software, we can poll several peripheral devices and transfer data when each is ready.

Interrupts

The interrupt I/O is a process of data transfer where-by an external device or a peripheral can inform the microprocessor that it is ready for communication.

Some pins on the 8085 allow peripheral equipment to interrupt the main program for I/O operations. When an interrupt occurs, the 8085 completes the instruction it is currently executing. Then it branches to a subroutine that services the peripheral device. Upon completion of the service subroutine, the CPU returns to the main program.

This type of I/O operation is called interrupt-driven I/O. It is most efficient than programmed I/O because the CPU does not wait for a high status signal. Instead, the CPU can process data while the peripheral device is getting ready for an I/O transfer.

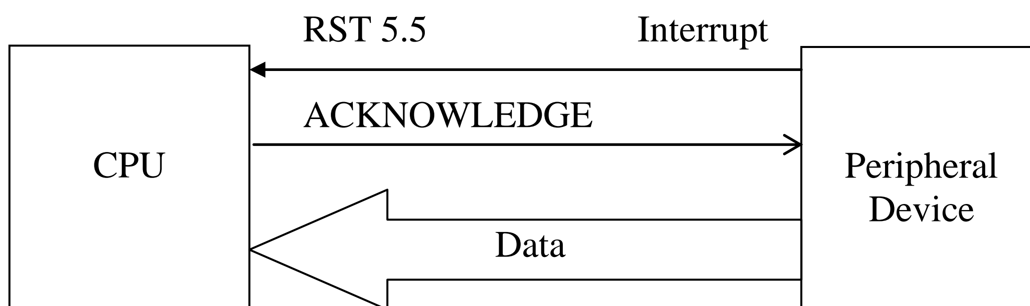
Hardware Restarts

RST 0 to RST 7 are software restarts because they are instructions. Besides these software restarts, the 8085 has for hardware restarts designated TRAP (pin 6), RST 7.5 (pin 7), RST 6.5 (pin 8) and RST 5.5 (pin 9). When any of these pins is active, the internal circuits of the 8085 produce a hardware call to a predetermined vector location. For example the RST 5.5 sends the program to 002ch.

A hardware call like this is know a vectored interrupt because the program branches to a vector location where the starting address of a service subroutine is stored. By connecting the hardware restart pins to peripheral devices, we can use interrupt-driven I/O instead of programmed I/O.

Interrupt-Driven I/O

The following figure illustrates the basic idea behind Interrupt-Driven I/O. When the peripheral device has a byte ready for data transfer, it sends a high bit to the RST 5.5 input. After saving the contents of the program counter in the stack, the CPU branches to vector location 002ch. Here it finds the starting address of the service subroutine; this subroutine inputs a byte from the peripheral device and stores it in memory.



After the byte has been stored, the CPU sends a high ACKNOWLEDGE bit to the peripheral device. This tells the peripheral device to get the next byte ready for transfer. Then the CPU returns to the main program where data processing continues. In this way, the CPU can process data in the main program rather than waiting for the peripheral device to get ready. When the peripheral device has the next byte ready for transfer, it sends a high bit to the RST 5.5 input and the cycle repeats.

The advantage of interrupt-driven I/O is its efficiency. The CPU is no longer wasting 99.9 percent of its time waiting for the peripheral device to set up the next byte. The typical microcomputer uses interrupt-driven I/O because it has to process data while servicing a keyboard, a video display, and other peripheral devices.

Interrupt Priorites

Besides the hardware restarts, the 8085 also has an INTR interrupt. The following table summarizes the 8085 interrupt. Notice that TRAP has the highest priority. RST 7.5 next highest, and so on. If two or more interrupts are active at the same time, the 8085 takes them in order to their priority level: TRAP is serviced first, then RST 7.5, and so forth.

Interrupt	Priority	Vector Location
TRAP	1	0024H
RST 7.5	2	003CH
RST 6.5	3	0034H
RST 5.5	4	002CH
INTR	5	NONE

Maskable Interrupt

The interrupt requests are classified in two categories:

1. Maskable interrupt request can be ignored or delayed by the microprocessor and used in telephone
2. Non - Mask able interrupt request the microprocessor respond immediately and used in smoke detector.

The RST 7.5, RST 6.5, and RST 5.5 interrupts are maskable, this means that they can be disabled by applying high M7.5, M6.5, and M5.5 signals. The TRAP is non-maskable; once it goes high and stays high a TRAP interrupt appears at the final output. The signals I7.5, I6.5, and I5.5 are called pending interrupts. The signal IE is called the interrupt enable flag.

To activate the RST 5.5 interrupt, I5.5 must be high, M5.5 must be low, and IE must be high.

Interrupt Instructions

Certain instructions are used with the interrupt. For instance, we might want to disable the interrupt system, or mask a particular interrupt, or examine pending interrupts, and so forth.

No.	Instruction	Type	No. of Bytes	Function	Effect
1.	DI	Mach. control	1	Disable Int.	None
2.	EI	Mach. control	1	Enable Int.	None
3.	RIM	Mach. control	1	Read Int. Mask	None
4.	SIM	Mach. control	1	Set Int. Mask	None

EI and DI

The 8085 has two instructions that can enable or disable all inturrupts except the TRAP. The instruction: **EI** stands for enable interrupt. When executed, this instruction will produce a high EI bit. This produce a high IE output.

The instruction: **DI** stands for disable interrupts This rusults in a low IE. The low IE then disables all interrupts except TRAP.

Because the interrupts are automatically disabled by the any interrupt acknowledge bit, the programmer usually includes an EI as the next last instruction in the service subroutine. For instance, the last two instructions typically are:

```
Subroutine: -----  ┐
            -----  │  I.S.R.
            EI     │
            RET    ┘
```

This subroutine cannot be interrupted (except by a TRAP) after the EI is exexuted, the processing returnes to the main program with the interrupt system enabled.

The programmer who wants some critical part of the main program to run uninterrupted can use a DI at the beginning of the segment to be protected and EI at the end:

```
Main program: DI
              ….
              ….
              EI
```

This protects the program between the DI and EI because the interrupt system is disabled.

SIM

(Set Interrupt Mask) This is a multipurpose instruction and used to implement
The 8085 interrupts 7.5, 6.5, 5.5, and serial data output. To use this instruction, you first load the accumulator as shown below:

				MSE	M7.5	M6.5	M5.5

Then by executing a SIM the accumulator will be transferred to the appropriate locations.

Here is an example. Suppose we want to mask (disable) the RST 7.5 and RST 6.5 interrupts and unmask (enable) the RST 5.5 interrupt. Then we can use:

```
MVI A,0eh
SIM
```

After the MVI is executed, you see a high MSE, high M7.5, and high M6.5; all other bits are low. The SIM then transfers these bits to the appropriate locations. This will preventing interrupts I7.5 and I6.5 from arriving at the final ouputs.

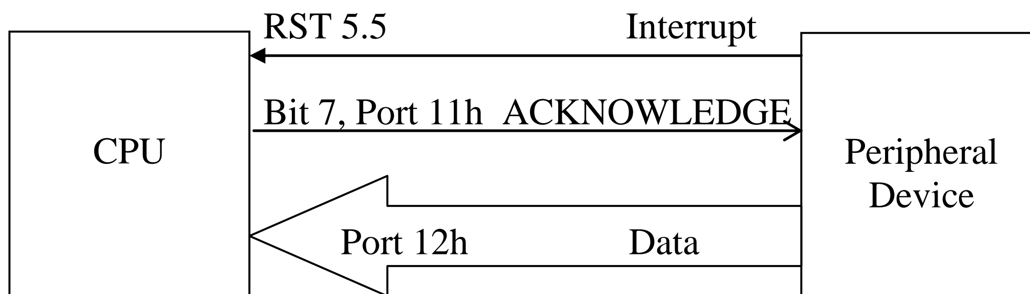
RIM

(Read Interrupt Mask) this is a multipurpose instruction used to read the status of interrupts 7.5, 6.5, 5.5 and read serial data input bit. When executed, it loads the accumulator. Bit 7 is the serial input data (SID), Bit 6, 5, and 4 are the pending interrupts. Bit 3 is the interrupt-enable bit IE. Bits 2,1, and 0 are the interrupt masks. Execution of a RIM allows the programmer to examine the status of the pendin interrupts, masks, and the like. This may be necessary following an interrupt service subroutine.

Example:

The following figure shows a peripheral device connected to the RST 5.5 interrupt. After the CPU recives a data word in port 12h, it can send a high ACKNOWLEDGE bit (bit 7 of port 11h) back to the peripheral device.

The starting address in the RST 5.5 vector location is F100h. Show a service subroutine that inputs data from the peripheral device and stores the data at 3000h



Solution

RST 5.5 Vectored Location

Address	**machine code**	
002c	c3	
002d	00	jmp to I.S.R (vectored location of RST 5.5 contain
002e	f1	the start address of interrpt service routine)

Main Program

Address	**Mnemonic**
2000h	MVI A,0eh
2002h	SIM
2003h	LXI SP,0ff00h ; set start address of the stack

	----
	aa: jmp aa

Or Main Program

```
Address     Mnemonic
2000h       MVI  A,0eh
2002h       SIM
2003h       LXI SP,0ff00h ; set start address of the stack
        aa: ----
            ---- in special case HLT
            ----
            jmp aa
```

Interrupt Service Routine (I.S.R)

Address	Mnemonic	Comment
F100h	PUSH PSW	; save accumulator and flags
F101h	PUSH H	; save HL contents
F102h	IN 12H	; Input data from device
F104h	LXI H,3000h	; set pointer
F107h	MOV M,A	; store data
F108h	MVI A,80H	; set ACKNOWLEDGE bit
F10Ah	OUT 11H	; Acknowledge data arrival
F10Ch	POP H	; restore HL contents
F10Dh	POP PSW	; restore accumulator and flags
F10Eh	EI	; enable interrupts
F10Fh	RET	; return

When the peripheral device is ready for data tranfer, it sends a high bit to the RST 5.5 input. After the 8085 recognizes this interrupt, it branches to vector location 002ch. Here it finds a jmp f100h. The jump takes the program to the starting address of the service subroutine.

The service subroutine usually destroys the contents of the accumulator and HL register. For this reason, the subroutine starts with a PUSH PSW and a PUSH H; this saves the accumulator contents, flags, and HL contents in the stack.

Next, the IN 12h inputs a data word from port 12h. After the HL pointer is set to 3000h, the data is stored at location 3000h. The next two instructions send a high ACKNOWLEDGE bit to the peripheral device.

The POP H and POP PSW restore the contents of the HL register, accumulator, and flag register. Because the stack operates as first-in last-out memory, we pop in the reserve order that we pushed.

Finally comes the EI to enable the interrupts and the RET to get us back to the main program.

BY modifiying this subroutine, we can store bytes in successive memory locations. For instance, using an INX H and some other instructions, we can update the HL pointer each time the subroutine is called. In this way, the incoming data words will be stored at 3000h, 3001h, 3002h, and so on.

Direct-Memory Acsees

A floppy disk is a thin plastic disk about 8 inches in diameter, coated with magnetic oxide. A disk drive is a peripheral device that can either read or write data on the disk, which can store a half million or more bytes. The only practical way to transfer data to and from the disk is with direct memory access (DMA). The 8085 can turn over control of its buses to a DMA controller for high-speed I/O transfers. In this way, large amounts of data can be transferred in a relatively short time.

Accumulator in the middle

The details of DMA transfer are too complicated to go into here, but we can discuss the basic idea. The IN instruction is the usual way to input data from peripheral devices. The accumulator is involved because it receives the input data. Similarly, the OUT instruction transfers data from accumulator to output devices. In either case, the accumulator serves as go-between.

On way to transfer data from the memory to peripheral devices is to use move and I/O instructions. For instance, to move 256 bytes from memory to an output device, we can use a loop that includes MOV A, M and OUT instructions. This approach will work, but it is too slow when large amounts of data are involved.

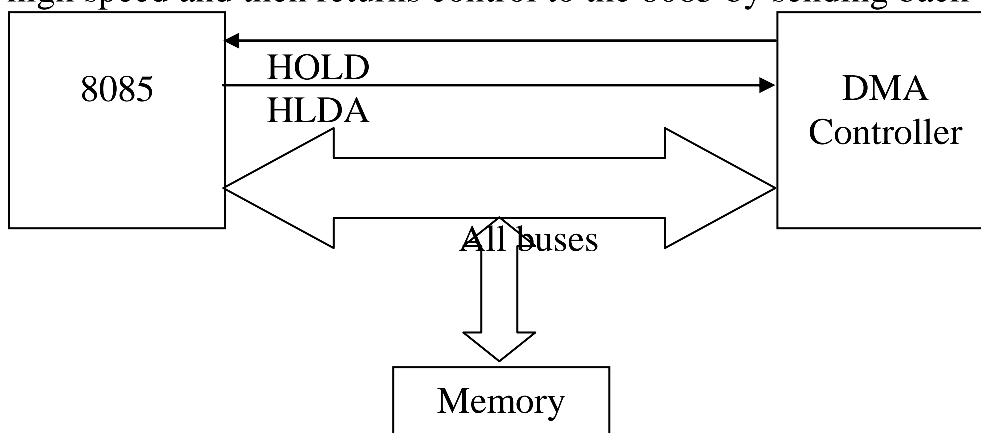
The problem

The foregoing approach is slow for two reasons. First, the accumulator acts as a halfway station in each transfer of data from memory to I/O, or vice versa. Second, the 8085 is micro programmed, which means that the microinstructions have to be read from a control ROM. The access time of this control ROM slows things down.

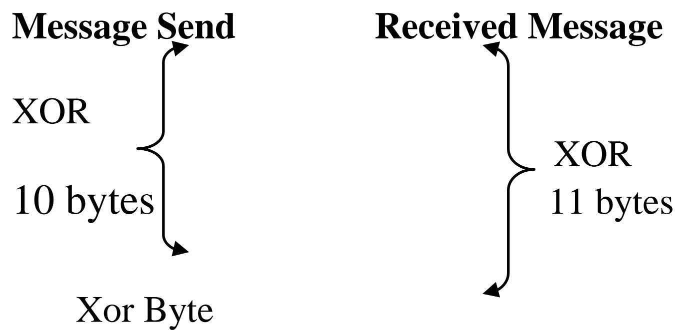
Basic Idea

DMA data transfers are faster because the accumulator is eliminated as a halfway station; the data goes directly from the memory to the peripheral device or vice versa. Also, the DMA controller has hardwired control instead of micro programming. This eliminates the access time of the control ROM.

The HOLD and HLDA signals are used in DMA operations. In the following figure, when the DMA controller is ready to take over control, it sends a high HOLD signal to the 8085. The 8085 then three-states (floats) its address, data, and control buses. It also sends a high HLDA (hold acknowledge) to the DMA controller, indicating that it has turned over control. The DMA controller carries out the data transfers at a high speed and then returns control to the 8085 by sending back a low HOLD signal.



Error Detection



Class Work

1. Detect the validity of received message (11 bytes read via port 72) after checking the availability of data via port 73 (status bit: 3). the validity is checked by calculate its checksum (Xor all received bytes) if the message is valid (checksum=0) then send Ack: 55 via port 74 otherwise send Nack: 15 via the same port. (Save the message in memory started at address 2080).

Address	HexCode	Label	Opcode	Operands	Comments
2000 2001 2002	 80 20		LXI	H,2080	; HL=2080
2003 2004 2005	 0B 00		LXI	B,00B	; BC=00B
2006 2007	 73	BYTE:	IN	73	; A←PORT 73
2008 2009	 08		ANI	8	; A=A AND 8
200A 200B 200C	 06 20		JZ	BYTE	; IF Z=1 then PC=2006
200D 200E	 72		IN	72	; A←PORT 72
200F			MOV	M,A	; M_{HL}=A
2010			INX	H	; HL=HL+1
2011			XRA	B	; A=A XOR B
2012			MOV	B,A	; B=A
2013			DCR	C	; C=C-1
2014 2015 2016	 06 20		JNZ	BYTE	; IF Z=0 then PC=2006
2017			MOV	A,B	; A=B
2018			ANA	A	; A=A AND A
2019 201A 201B	 21 20		JZ	ACK	; IF Z=1 then PC=2021
201C 201D	 15		MVI	A,15	; A=15
201E 201F	 74		OUT	74	; PORT 74←A
2020			RST1		; END
2021 2022	 55	ACK:	MVI	A,55	; A=55
2023 2024	 74		OUT	74	; PORT 74←A
2025			RST1		; END

2. Update the previous program by using I.S.R. (Request Line 6.5) to receive the message and set register D to 1 when calculate the checksum, the main program detect if register D=1 then check the value of checksum if it is 1 then send ACK otherwise send NACK.

Address	HexCode	Label	Opcode	Operands	Comments
2000 2001 2002	 50 20		LXI	SP,2050	; SP=2050
2003 2004 2005	 0B 00		LXI	B,00B	; BC=00B
2006 2007	 00		MVI	D,0	; D=0
2008 2009 200A	 80 20		LXI	H,2080	; HL=2080
200B 200C	 0D		MVI	A,0D	; A=0D
200D			SIM		; SIM
200E		DINT:	MOV	A,D	; A=D
200F			ANA	A	; A= A AND A
2010 2011 2012	 0E 20		JZ	DINT	; IF Z=1 then PC=200E
2013 2014	 0F		MVI	A,0F	; A=0F
2015			SIM		; SIM
2016			MOV	A,B	; A=B
2017			ANA	A	; A= A AND A
2018 2019 201A	 20 20		JZ	ACK	; IF Z=1 then PC=2020
201B 201C	 15		MVI	A,15	; A=15
201D 201E	 74		OUT	74	; PORT 74 ←A
201F			RST1		; END
2020 2021	 55	ACK:	MVI	A,55	; A=55
2022 2023	 74		OUT	74	; PORT 74 ←A
2024			RST1		; END
2025 2026	 72	CHKINT:	IN	72	; A←PORT 72
2027			MOV	M,A	; M_{HL}=A
2028			INX	H	; HL=HL+1
2029			XRA	B	; A=A XOR B
202A			MOV	B,A	; B=A
202B			DCR	C	; C=C-1
202C 202D 202E	 31 20		JNZ	EXIT	; IF Z=0 then PC=2031
202F 2030	 01		MVI	D,1	; D=1
2031		EXIT:	EI		; EI
2032			RET		; PC=ADDRESS AFTER INT. HAPPEND

RST 6.5

0034: C3
0035: 25
0036: 20
} JMP CHKINT