

HAND-HELD REMOTE KEYPAD FOR IBM PC-AT

PROJECT REPORT



Submitted by

G.RAJAJI

K.RAJESH KANNAN

N.UDAYA KUMAR

T.N.C.VENKATA RANGAN

GUIDE

Mr.C.S.MOHAN RAM, M.E

Lecturer

Department of Electronics and Communication Engineering.

*In partial fulfilment of the requirements for the
award of Degree of Bachelor of Engineering*



Department of Electronics and Communication Engineering.

SRI VENKATESWARA COLLEGE OF ENGINEERING

PENNALUR, SRIPERUMBUDUR - 602 105.

MARCH 1996

Department of Electronics and Communication Engineering.

SRI VENKATESWARA COLLEGE OF ENGINEERING

Pennalur, Sriperumbudur - 602 105



CERTIFICATE

This is to certify that the project "HAND-HELD REMOTE KEYPAD FOR IBM PC-AT" is a bonafide work done at the Department of Electronics and Communication Engineering, SRI VENKATESWARA COLLEGE OF ENGINEERING, Pennalur, Sriperumbudur - 602 105 during the academic year 1995 - 96, by

G. RAJAJI

(Reg. No. 822 590)

K. RAJESH KANNAN

(Reg. No. 822 591)

N. UDAYA KUMAR

(Reg. No. 822 644)

T. N. C. VENKATA RANGAN (Reg. No. 822 647)

and submitted to the UNIVERSITY OF MADRAS in partial fulfilment of the requirements for the award of degree of Bachelor of Engineering.

Faculty Guide

Head of the Department

Submitted to the University Examination held on March '96

Internal Examiner

External Examiner

I had occasion to interact with the team members doing a project titled, "**Hand-held Remote Key Pad for the IBM PC-AT**", namely

G Rajaji

K Rajesh Kannan

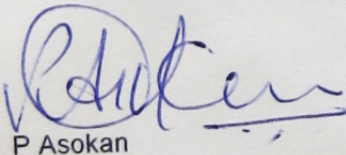
N Udaya Kumar

T N C Venkata Rangan

I found the idea very innovative and the design well conceived. The prototype of the project was also demonstrated to me and it worked well. I believe that this project can evolve into a commercially viable a product. I recommended the idea and the team to an independent designer, who is making wheel alignment systems. He also expressed a great deal of appreciation for the idea and has commended the capability of the team that thought of and executed this project. He has also indicated that he is considering making this a part of his product.

My interactions with the team have been quite interesting and I have no doubt that the team members will be an asset to the nation. I have no doubt in asserting that the team can be quite proud of their achievement and I am sure the college and the teachers share my pride and also deserve kudos for supporting such wonderful efforts on the part of the students.

I wish all of them great success in their careers.



P Asokan

Director - Development

ACKNOWLEDGEMENTS

From the bottom of our heart we express our gratitude and thanks to **Mr. R.Arvind**, Freelance Inventor who supported us morally during our rough times and to **Mr. Rajagopal P.Sarathy** of Binghamton, U.S.A, who sent us the components from abroad.

We also thank our Guide, **Mr. C.S. Mohan Ram**, Lecturer, for his support time and again.

Our grateful thanks are also felt to **Prof. R. Ramachandran**, Head of our Department and the entire teaching and Non-teaching staff of our department, who have been with us always as a constant source of inspiration for our success.

CONTENTS

☞ Synopsis	1
☞ Design Goals	2
☞ Infra Red Remote Control Basics	3
☞ Development Platform	7
☞ Technical Requirement s	8
☞ Hardware: Hand-Held Remote Keypad	9
☞ Inside Out	13
☞ Software: Hand-Held Remote Keypad	47
☞ Problems Faced	54
☞ Applications	55
☞ Bibliography	56
☞ Appendix A : IR Sensor Technical Specifications	
☞ Appendix B : Philips Remote Transmitter Specifications	
☞ Appendix C : Photometric Terminologies	
☞ Appendix D : How The 8086 Uses I/O Ports	
☞ Appendix E : Interrupt And 80x86 Family	

SYNOPSIS

Computers are now in our living rooms, the familiar sight here is a remote control used for Audio and Video Systems, and our project idea stems from here. Even when speech and character recognition are being researched, the primary interface to Computers still continues to be the familiar desktop keyboards.

Our project aims at using such a “Remote control” as a keypad for PC-AT desktops. The project includes research and development of hardware and the supporting software on MS-DOS® platform.

The “Remote KeyPad” is aimed at providing a mobile interface between the user and the computer/controlled device, in addition to the conventional desktop keyboard and not a substitute for it.

The “Remote KeyPad” outshines conventional keyboard due to its ability to be carried around in a confined area and still providing access to the computer. Its compact and handy design is striking. It is mandatory for a layman who sees Computers in a different perspective from that of the developers. It may be a boon to make him work efficiently and conveniently, as he is already familiar with the remote.

DESIGN GOALS

- ☑ The KeyPad in its initial form should use an existing Remote Control transmitter (used in TV/VCR/CATV), this is due to the time constraints and the cost involved,
- ☑ It should be relatively simple to implement and duplicate if needed in mass numbers,
- ☑ It should not implement its own dedicated interface, which complicates installation and usage,
- ☑ It should use a standard interface to the computer (PC-AT) like the Serial/Parallel ports available in the PC . This also facilitates a wider ranges of compatibility and also cost-effective,
- ☑ Since PC-AT is a very sensitive hardware orchestra, any hardware which we connect to it, should be designed carefully to meet the standards of PC,
- ☑ The software and the product should be very easy to be used by layman also,
- ☑ The “KeyPad” should work with already developed applications, that is it should be backward compatible,
- ☑ The applications must be totally unaware of “KeyPad”, they should see the output of “KeyPad” as normal desktop keyboard scan codes,
- ☑ The KeyPad may provide a specialised interface to applications which are “KeyPad” aware, eliminating the BIOS/OS layer for greater and speedier functionality,
- ☑ The “KeyPad” key mapping may be allowed to be changed/configured by the user.
- ☑ The software must work in DOS platform and may be extended to MS-Windows™.

INFRA-RED REMOTE-CONTROL BASICS

Any remote control system basically consists of a transmitter and a receiver. An Infra-red system can be easily be designed to give very effective single- or multichannel remote-control operation: Here, the hand-held control unit transmits a coded waveform via a broad infra-red beam, and this signal is detected and decoded in the remotely-placed receiver and hence used to activate external devices, etc., via the receiver outputs.

The transmitter can remote-control a receiver that is placed anywhere within the active area of the IR beam, and that the effective operation must be in line-of-sight contact; also note that an object placed within the beam can create a blind area in which line-of-sight contact cannot exist.

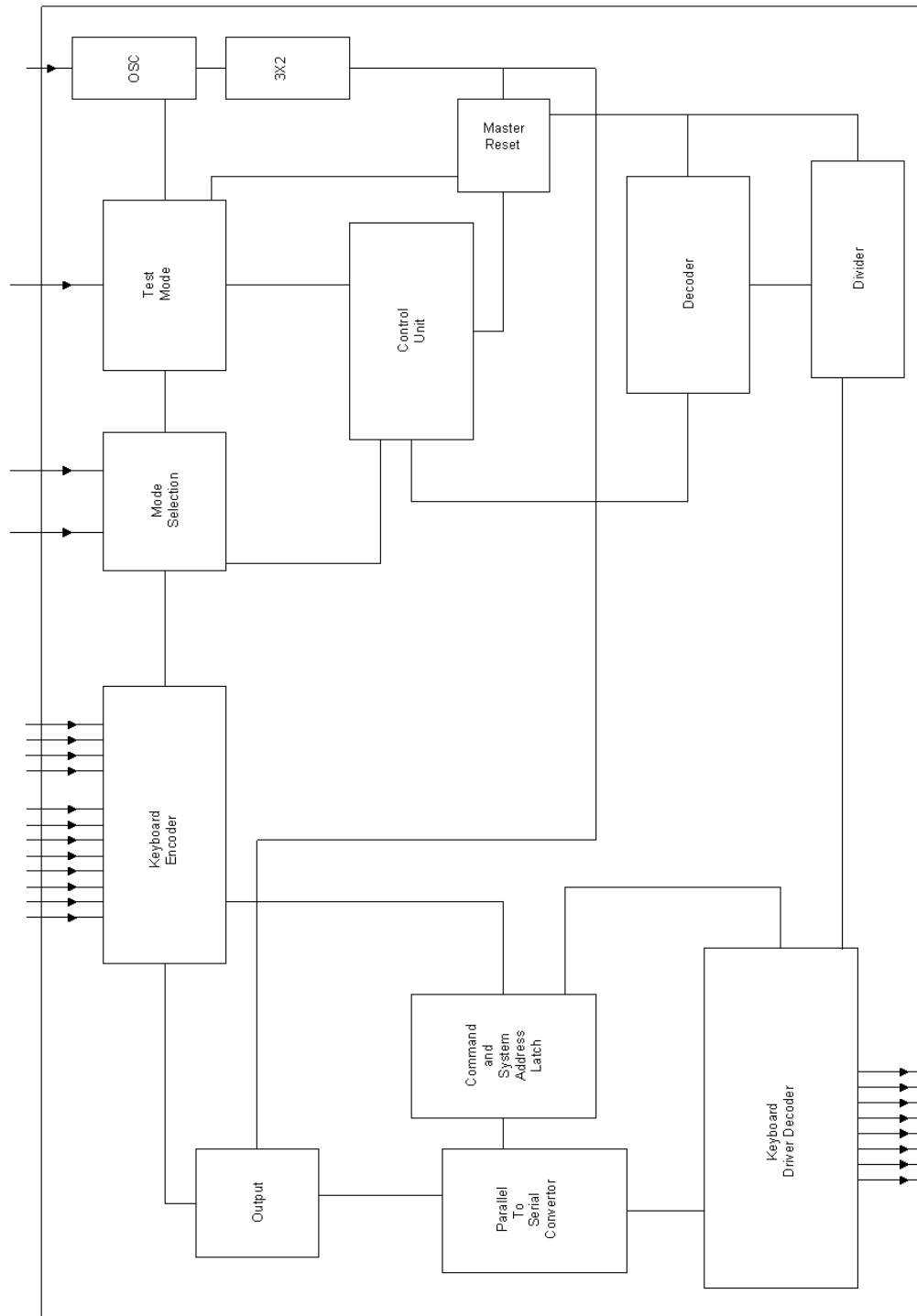
CODE PATTERNS

Most modern IR remote-control systems give multi-channel operation, with each channel giving digital control of an individual function. The transmitter waveforms usually take a predetermined form, for example, a burst of six bits of a standard digital format. Generally, it consists of a frame of fixed time period with a synchronisation bit. In most of the implementations, the designers have felt it rudimentary to implement a stop bit.

BLOCK DIAGRAM

The following diagram shows the typical block diagram of a multichannel IR remote control transmitter. This unit is usually fitted with a multifunction keyboard, which has its X and Y outputs repeatedly scanned via an encoder circuit, that controls the input to a code waveform generator system. This latter unit, generates the carrier wave signal and a fixed number of bits plus sync pulse (repeating frame waveforms), which are then passed on to a standard infra-red transmitter output stage .

Fig. 1. Block Diagram of a typical Remote Control Transmitter



In the receiver circuit, the detected IR signal is first fed to a fairly sophisticated pre-amplifier stage which provides very high gain for long range operation but does not saturate if the transmitter is used near the receiver.

The pre-amplifier output is fed to an LSI decoder IC, which typically directly provides three or four digital outputs (simple on/off functions) and two or three analogue outputs, but also provides a fixed number of bits as output, that is a parallel-coded version of the original serial code and can optionally be decoded via additional ICs to give more functions.

PRACTICAL SYSTEMS

Simple single-channel IR remote-control systems can easily be built using discrete components such as bipolar transistors. Multichannel system with up to six digital channels are only slightly more complicated, and can be built with the aid of simple CMOS ICs such as the 4017B, etc.

In early 80s, if more than six channels are required, it is best to use dedicated LSI remote control ICs for the purpose. Several manufacturers produces dedicated ICs of this type, the best known of these being the 490/922 thirty-two-channel range of devices from Plessey, and the IR60 sixty-channel range of devices from Siemens,

Recently, Sharp and Siemens have introduced high efficiency, low cost, high gain sensors like SFH505A, SFH506-36, IS1U60. Philips Semiconductors are ahead with their transmitter and receiver IC ranges in this field with the ICs like SAA3010, TDA3048 and SAA3049.

PHILIPS AND SONY RC-5 CODING SYSTEM:

This is one of the most famous coding techniques used in modern day remote control systems. The pioneers for this coding are Philips and Sony

who use it extensively in their range of products. The main advantages of this system are :

1. stable and well defined coding system.
2. supports a large number of commands (2048 commands).
3. commands are arranged in easily addressable groups.
4. system is immune to sunlight, incandescent lamps and fluorescent lamps.
5. avoiding of collisions between multiple installations.
6. individual identification for each type of transmitter.
7. the 36-kHz carrier frequency used is immune from TV line scan interference.
8. acceptable distance of operation.

SANYO CODING SYSTEM:

Even though this is a proprietary coding system, it is used extensively in many Sanyo appliances. In this technique, several advantages are inherited, from RC-5 coding system, but the complexities are reduced.

The command consists of 16 bits of data sent at a frequency of 1000Hz, with a duty cycle of 50% to represent 20-odd keys. In its present format, the code does not support multichannel operation. Advanced versions of this code are implemented in latest systems.

DEVELOPMENT PLATFORM

Study Equipment

- Scientific Storage Oscilloscope 60Mhz

Machine

- 486DX-2, 66Mhz,
- 12MB RAM,
- MS-DOS® 6.22
- MS-Windows™ for Workgroups 3.11
- HP-Deskjet 520

Tools

- Turbo C 2.0 for MS-DOS®
- MS-Visual C++ 1.5
- MASM 6.0 for MS-DOS®

TECHNICAL REQUIREMENTS (for target platform)

MINIMAL REQUIREMENTS

- IBM PC-AT or Compatibles and IBM PS/2
- CGA or Better equipped video adapter
- Monochrome Monitor
- 8250-UART
- RS-232C interface with DB-9 connector
- MS-DOS® 3.3
- 101-Keybaord

RECOMMENDED REQUIREMENTS

- ✓ PC-AT with 386 or better processor
- ✓ VGA, 640x480, 16 Colors, Video adapter
- ✓ 16550A-UART with 16 byte buffer
- ✓ MS-DOS® 5.0 or better
- ✓ MS-Natural Keyboard™
- ✓ Enhanced Parallel Port for controlling external devices.
- ✓ ISA 8/16 bit slot (for internal version only).

HARDWARE: HAND-HELD REMOTE KEYPAD

Our system uses Sanyo Coding System. The choice between RC-5 and Sanyo Coding system was not based on any technical superiority of the latter but due to simplicity involved and the availability of components.

The transmitter we have employed is an Universal Remote Control Transmitter. It supports Sanyo Coding System along with several other coding systems. The advantages of this transmitter:

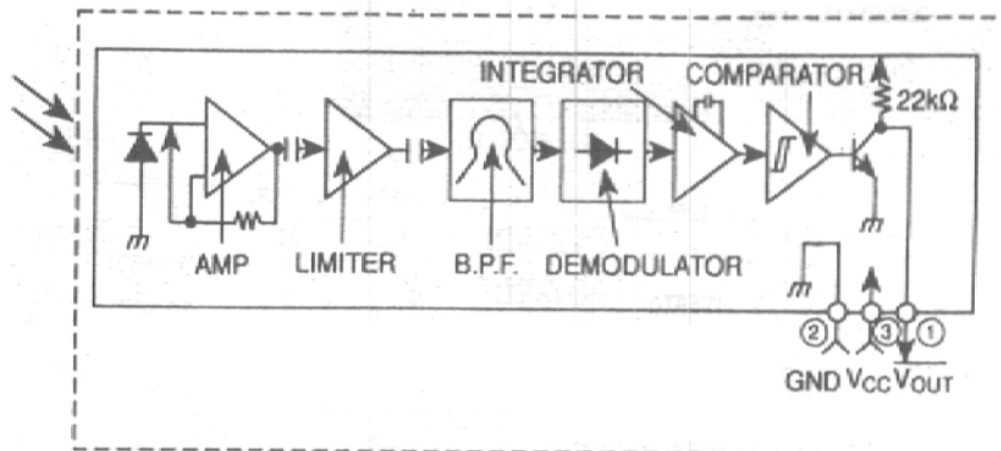
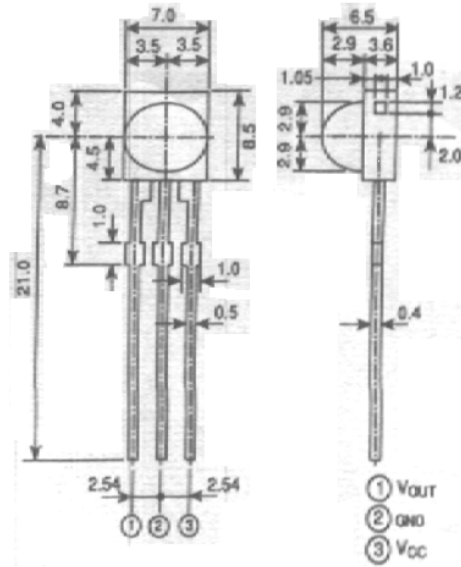
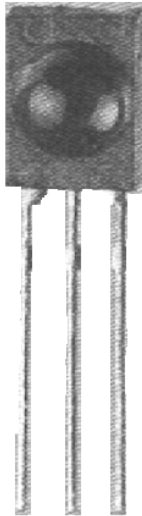
- Easy availability.
- Economical for small requirements
- Compact design.
- Availability of more number of keys.
- Robust design.
- Extended Battery life.

Transmitters like the newly introduced Philips OEM Remote-Control Units can also be used with due modifications in the receiver software. See Appendix for details on this System.

The receiver unit we have implemented consists of an Infrared Sensor (IS1U60). This has a built in pre-amplifier and a demodulator. This is a high gain, low noise receiver and is capable of detecting signals without appreciable directional properties.

The sensor is stable in its operation, primarily due to its robust design. Our experience with this IC stands a proof to this claim by the manufacturer. (The IC was even kind enough to forgive our mischief with it in handling and testing.) The technical specifications can be referred from the Appendix.

SHARP IS1U60 DETAILS



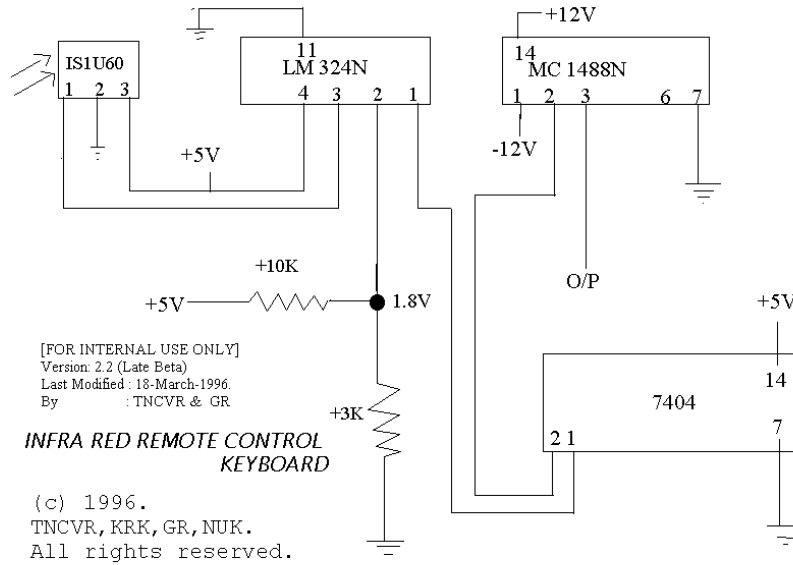
The detected signal from the sensor is fed to a voltage comparator (LM324), which produces an output at TTL levels. The comparator's high input impedance offers or draws very little current so that the sensor is not loaded beyond its capabilities.

The output of the comparator is capable of meeting to high current requirements. Since we are interested in detecting this signal by the PCAT through the available RS-232C interface, we convert the output of the comparator. The conversion is effected due to the two different operating voltage levels. A TTL to EIA converter device (1488) is employed for this purpose.

The EIA output is fed into the IBM PC AT compatibles through a RS 232C DB-9 female connector. For our purpose, we employed the Transmit and GND pins of this interface.

The serial port reads this received signal at twice its rated frequency of 1000Hz. This is processed by the accompanied software.

An alternative approach to the above hardware design developed by us is by the use of dedicated decoders already available with OEMs. This approach has the advantage of parallel outputs of the serial input. This can then be read through any parallel interface with the computer (like the Centronics Port in PC).



INSIDE OUT

The design of hardware was the first stage in the development of “KeyPad”. Before we could design the software for the “KeyPad”, we had to thoroughly understand PC’s Asynchronous Communication adapter, Serial Port and interrupt handling. It also becomes important to look into the keyboard handling in PC, Direct Video mapping techniques and MS-DOS’s TSR programming concepts and Interrupt Service Routines.

The topics discussed are as follows:

- ▢ IBM Asynchronous Communications Adapter
- ▢ Serial Port
- ▢ 8250 I/O Port
- ▢ Keyboard Operation
- ▢ Interrupt Handlers
- ▢ Memory Mapped Display Techniques
- ▢ Using The System Timer Tick

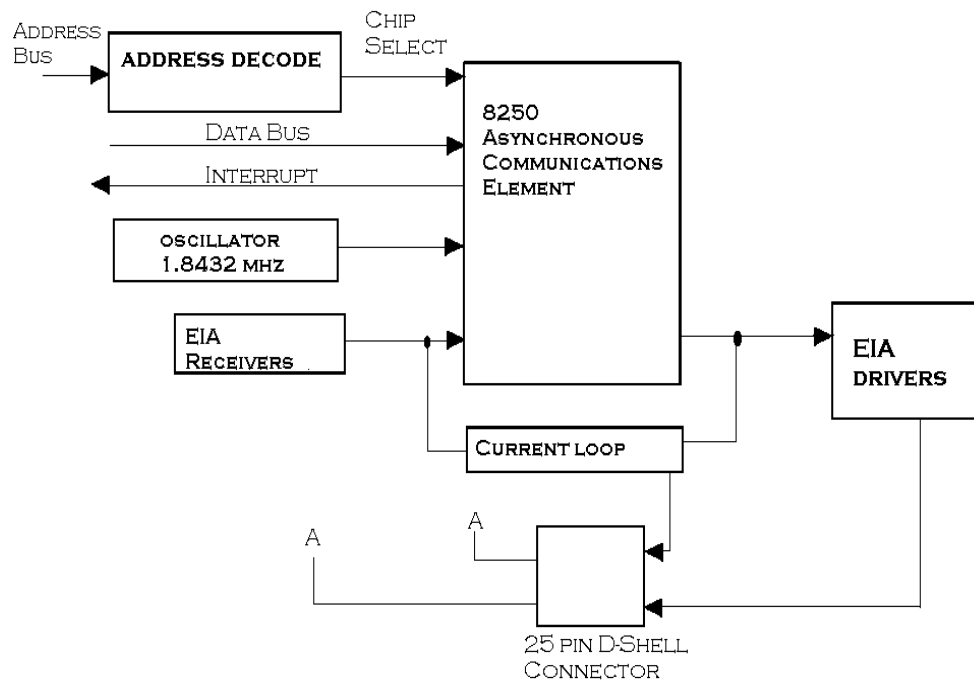
IBM ASYNCHRONOUS COMMUNICATIONS ADAPTER

The adapter is fully programmable and supports asynchronous communications only. It will add and remove start bits, stop bits, and parity bits, a programmable baud rate generator allows operations from 50 baud to 9600 baud. Five, six, seven or eight bit characters with 1,1-1/2, or 2 stop bits are supported. A fully prioritised interrupt system controls transmit, receive, error, line status, and data set interrupts.

The heart of the adapter is a INS8250 LSI chip or functional equivalent. features in addition to those listed above are:

1. Full double buffering eliminates need for precise synchronisation .
2. Independent receiver clock input.
3. Modem control functions:
 - 3.1. clear to send (CTS),
 - 3.2. request to send (RTS),
 - 3.3. data set ready (DSR),
 - 3.4. data terminal ready (DTR),
 - 3.5. ring indicator (RI) and
 - 3.6. carrier detect.

BLOCK DIAGRAM OF THE ADAPTER



MODES OF OPERATION:

The different modes of operations are selected by programming the 8250 asynchronous communications element. This is done by selecting the I/O address (hex 3F8 to 3FF primary, and hex 2F8 to 2FF secondary) and writing data out to the card. Address bits A0, A1, and A2 select the different registers that define the modes of operation. Also, the divisor latch access bit (bit 7) of the line control register is used to select certain registers.

INTERRUPTS:

One interrupt line is provided to the system. This interrupt is IRQ4 for a primary adapter or IRQ3 for an alternate adapter, and is positive active to allow the communications card to send interrupts to the system, bit 3 of the modem control register must be set to 1(high). At this point, any interrupts allowed by the interrupts enable register will cause an interrupt.

The data format is as follows:

Data bit 0 is the first bit to be transmitted or received. The adapter automatically inserts the start bit, the correct parity bit if programmed to do so, and the stop bit (1,1-1/2,or,2 depending on the command in the line-control register).

INTERFACE DESCRIPTION

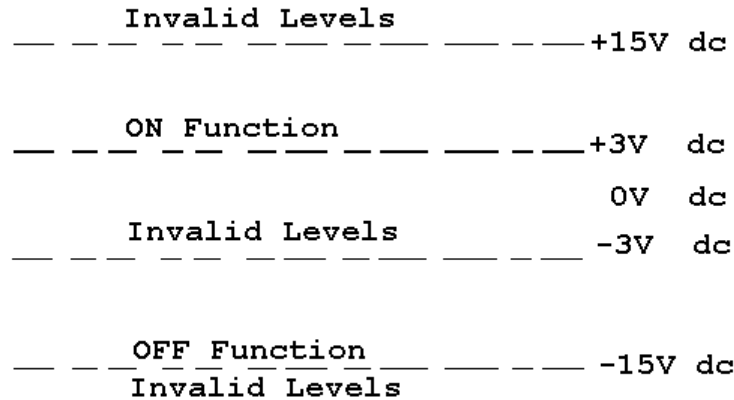
The communications adapter provides an EIA RS-232C-like interface. Once 25-pin D-shell, male type connector is provided to attach various peripheral devices. The voltage interface is a serial interface. It supports certain data control signals, as listed below.

Pin 2	Transmitted Data
Pin 3	Received Data
Pin 4	Request to Send
Pin 5	Clear to Send
Pin 6	Data Set Ready
Pin 7	Signal Ground
Pin 8	Carrier Detect
Pin 20	Data Terminal Ready
Pin 22	Ring Indicator

These signals are sampled or generated by the communications control chip. These signals can then be sensed by the software to determine the state of the interface or peripheral device.

VOLTAGE INTERCHANGE INFORMATION:

Interchange Voltage	Binary State	Signal condition	Interface Condition
Positive Voltage =	Binary (0)	= Spacing	= On
Negative Voltage =	Binary (1)	= Marking	= Off



The signal will be considered in the “marking” condition when the voltage on the interchange circuit, measured at the interface point, is more negative than -3Vdc with respect to signal ground. The signal will be considered in the “spacing” condition when the voltage is more positive than +3Vdc with respect to signal ground. The region between +3Vdc and -3Vdc is defined as the transition region, and considered an invalid level. The voltage that is more negative than -15Vdc or more positive than +15Vdc will also be considered an invalid level.

During the transmission of data, the “marking” condition will be used to denote the binary state “1” and, “spacing” condition will be used to denote the binary state “0”.

For interface control circuits, the function is “on” when the voltage is more positive than +3Vdc with respect to signal ground and is “off” when the voltage is more negative than -3Vdc with respect to signal ground.

There is an alternative Asynchronous Adapter Connector called 9-pin RS-232C connector (connector male 9-pin D subminiature: cable requires female)

Pin Number	Signal Direction	Description
1	←	DCD (Data Carrier Detect)
2	←	RX (Receive Data)
3	→	TX (Transmit Data)
4	→	DTR (Data Terminal Ready)
5		GND (Signal Ground)
6	←	DSR (Data Set Ready)
7	→	RTS (Request To Send)
8	←	CTS (Clear To Send)
9	←	RI (Ring Indicator)

THE SERIAL PORT

The serial port is one of the most versatile interfaces in the PC. It can be connected to terminals, printers, plotters, mice, modems, other computers, and even certain disk drives. In most IBM PC compatibles, the serial interface is controlled by an 8250 or equivalent chip. The CPU communicates with this chip via eight I/O ports.

BIOS software interrupt 14h provides routines that control the serial port. MS-DOS uses these routines to implement the COM1 and COM2 devices. These routines check the line status register to see if a character has been received or if it is OK to send a character. This works fine at low speeds required by printers, but for high speed data communications it often loses data.

You can use interrupts to make sure that you do not lose any data. The first step is to set up the serial port to generate an interrupt when it receives a character. The Interrupt Service Routine needs to read the character from the serial port and save it in a buffer. This means that the main routine does not need to spend time checking for data at the serial port. It can do whatever processing is required, and get data from the buffer whenever it is ready.

MS-DOS support for serial ports is weak compared with its keyboard, video-display and printer support. This is one area where the application programmer is justified in making programs hardware dependent to extract adequate performance.

Programs that restrict themselves to MS-DOS functions to ensure portability can use the handle read and write functions, with the predefined standard auxiliary handle (3) to access the serial port. For example, the

following code writes the string 'hello' to the serial port that is currently defined as the Auxiliary device:

```
msg db 'hello' ; message for serial port
msglen equ $-msg ; length of message
...
...
mov ah,40h ; function 40h - write file or device
mov bx,3 ; BX - standard aux handle
mov cx,msglen ; CX - string length
mov dx,seg msg ; DS:DX - string address
mov ds,dx
mov dx,offset msg
int 21h ; transfer to MS-DOS
jc error ; jump if error
```

The standard auxiliary handle gives access to only the first serial port (COM1). If you want to read or write COM3 using the handle calls, you must issue an open request for the desired serial port and use the handle returned by that function with Int 21H Functions 3FH and 40H.

Some versions of MS-DOS have a bug in character-device handling that manifests itself as follows: If you issue a read request with Int 21H Function 3fH for the exact number of characters that are waiting in the drivers buffer, the length returned in the AX register is the number of characters transferred minus one. You can circumvent this problem by always requesting more characters than you expect to receive or by placing the device handle into binary mode using Int 21H Function 44H.

MS-DOS translates the traditional auxiliary-device functions into calls on the same device driver used by the handle calls. Therefore, it is generally preferable to use the handle functions in the first place, because

they allow very long strings to be read or written in one operation, they give access to serial ports other than COM1, and they are symmetrical with the handle video-display, keyboard, printer, and file I/O methods described elsewhere in this book.

Although the handle or traditional serial-port functions allow you to write programs that are portable to any machine running MS-DOS, they have a number of disadvantages, which are,

1. The built-in MS-DOS serial-port driver is slow and is not interrupt driven.
2. MS-DOS serial-port I/O is not buffered.
3. Determining the status of the auxiliary device requires a separate call to the IOCTL function if you request input and no characters are ready, your program will simply hang.
4. MS-DOS offers no standardised function to configure the serial port from within a program

A more flexible technique for serial-port I/O is to call the IBM ROM BIOS serial-port driver by means of Int 14H. You can use this driver to initialise the serial port to a desired configuration, and baud rate, examine the status of the controller, and read or write characters.

As with the ROM BIOS printer driver, the serial-port numbers used by the ROM BIOS are zero-based, whereas the serial-port numbers in MS-DOS logical-device names are one based. In this example, serial port 0 corresponds to COM 1.

Unfortunately, like the MS-DOS auxiliary-device driver, the ROM BIOS serial-port driver is not interrupt driven. Although it will support higher transfer speeds than the MS-DOS functions, at rates greater than 2400 baud it may still lose characters. Consequently, most programmers

writing high-performance applications that use a serial port (such as telecommunication programs) take complete control of the serial-port controller and provide their own interrupt driver. The built-in functions provided by MS-DOS, and by the ROM BIOS in the case of the IBM PC, are simply not adequate.

8250 I/O PORTS

Due to the limited support of serial ports in DOS and several other restrictions in using serial ports through ROM BIOS routines we have resorted to direct access techniques described in detail below.

INITIALISING THE SERIAL PORT

There are two steps to initialising the Serial Port for Data Communications with interrupts. First, you must tell the 8250 about the kind of data that it should use, and then set up the Interrupt Service Routine.

Receiver Buffer Register (DLAB=0)

Transmitter Holding Register (DLAB=0)

Divisor Low Byte (DLAB=1)

COM1: 3F8h

COM2: 2F8h

Bit 7-0	:	Data
---------	---	------

Interrupt Enable (DLAB=0)

Divisor High Byte (DLAB=1)

COM1: 3F9h

COM2: 2F9h

Bits 7-4	:	Reserved
Bit 3	:	Modem Status
Bit 2	:	Receiving Data
Bit 1	:	Transmit Ready
Bit 0	:	Data Available

INTERRUPT ID REGISTER

COM1: 3FAh

COM2: 2FAh

Bits 7-3	:	Reserved
Bit 2-1	:	Interrupt Type
Bit 0	:	No Interrupt

LINE CONTROL REGISTER**COM1: 3FBh**
COM2: 2FBh

Bits 7	:	DLAB
Bit 6	:	Send Break
Bit 5	:	Stick Parity
Bit 4	:	Even Parity
Bit 3	:	Parity Enable
Bit 2	:	Stop Bits
Bit 1-0	:	Character Size

MODEM CONTROL REGISTER**COM1: 3FCh**
COM2: 2FCh

Bits 7-5	:	Reserved
Bit 4	:	Lop Back
Bit 3	:	OUT2
Bit 2	:	OUT1
Bit 1	:	RTS
Bit 0	:	DTR

LINE STATUS REGISTER**COM1: 3FDh**
COM2: 2FDh

Bits 7	:	Reserved
Bit 6	:	Character Sent
Bit 5	:	Transmitter
Bit 4	:	Break Detect
Bit 3	:	Framing Error
Bit 2	:	Parity Error
Bit 1	:	Overrun Error

Bit 0	:	Data Ready
-------	---	------------

MODEM STATUS REGISTER

COM1: 3FEh
COM2: 2FEh

Bits 7	:	Carrier Detect
Bit 6	:	Ring Indicator
Bit 5	:	Data Set Ready
Bit 4	:	Clear to Send
Bit 3	:	Delta CD
Bit 2	:	Ring Indicator
Bit 1	:	Delta DSR
Bit 0	:	Delta CTS

First, we should set the Data parameters, i.e., the Number of Start, Stop, Parity and Data bits. Setting the divisor changes the frequency of the Clock signal. The clock signal marks the beginning and the end of each bit. The number of times the clock goes up or down in a second is the baud rate. The baud rate of the serial port must match the baud rate of the data it receives from the CPU.

The Line Control Register in the 8250 describes that the data that will be sent or received. Each part of the data signal is controlled by the Line Control Register. The data portion can be from five to eight bits long. Most data transfer programs use eight bits because this matches the size of a byte. Some text-only devices, like printers, use seven bits to match the size of an ASCII character.

After the data portion there may be a *parity* bit. The parity bit can be used to tell if the data portion is correct. Different devices, use the parity bit in different ways. Some devices do not use it at all (no parity), others

always set the parity bit (mark parity), and still others clear the parity bit (space parity). If the number of one-bits in the data (including the parity bit) is even, then the data has even parity. If the number is odd, then the data has odd parity. When the parity that the receiver (any device with a serial interface) expects is as different from the data received, then the receiver knows that the data is bad and should not be used.

The last part of a serial data byte is one or more stop bits. More serial devices use a single stop bit to mark the end of the data. Some older devices may use a single stop bit to mark the end of the data. Some older devices may use two stop bits or 1.5 stop bits. After the stop bit, the signal drops to the low voltage and stays there until the next data byte is sent.

The second half of the initialisation process is to set up the ISR and turn on serial interrupts. The first step is to make the entry for the serial port in the interrupt vector table point to the ISR. This must be done before any serial interrupts are possible, so that the processor will have some place to go when the interrupt occurs.

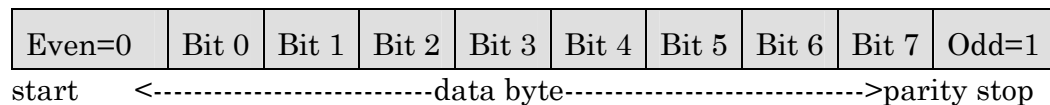
Next, you must tell the serial port to generate an interrupt when it receives data. To do this you must set the modem control register to enable the interrupt signal (OUT 2). Then tell the interrupt-enable register what situations should generate an interrupt. IN this case the 8250 will generate an interrupt when it has received a byte. The other three conditions are: ready to send a byte, data error detected, and modem status lines changed.

Finally, the 8259 must be told to allow interrupts from the serial port by clearing the bit for the serial interrupt in the interrupt mask register. Now the interrupts are all set up so you can set the interrupts-enabled bit with an STI instruction. Every time the serial port receives a character, it will call the ISR.

THE SERIAL FRAME

The Serial Frame consists of a combination of the start, data, parity, and stop bits. One frame at a time is sent or received on the serial link. For transmission, the UART adds the start bit, optional parity bit and stop bit to the data to be sent. On data reception, the UART checks the frame for various error conditions, and extracts the data from the frame.

The frame size depends on the various control options selected within the UART by software. The data portion can be from 5 to 8 bits long, a parity bit can be optionally included and from 1 to 2 stop bits can be selected. This means the total frame can range from 7 bits up to 12 bits long. The following figure shows a typical serial frame. The values shown are those measured at the UART chip, which are inverted from the serial port connector. The idle state is 1 at the UART when no frames are being sent.



SEQUENCE OF EVENTS-SERIAL RECEIVE

To receive a byte from the serial link, the port is assumed to be initialised with the proper baud rate and serial frame options. We will also assume that the byte comes from the serial port 3.

1. First the serial I/O port base address is determined by reading a word from the BIOS data area at 40:4h from serial port 3. If the value is zero, there is no active serial port attached, and no data can be received.
2. The modem control line Data Terminal Ready is set high. This informs the attached device that the computer is active and ready for communications. This is done by writing the value 1 to the Modem Control Port.

3. Next check Data Set Ready. This appears in the modem status register, bit 5. Data Set Ready indicates the attached device is powered up and ready. Data Set Ready should be checked until it becomes high, or up to 2ms before a time-out is signalled.
4. The receive buffer is next checked to see if data has been received. The Line Status Register, bit 0, holds a data ready flag. It is set to one when the receive buffer has data. If the data ready flag fails to get set after 2ms, then a time-out is signalled, and the operation aborted.
5. If no time-outs have occurred so far, the byte can be read from the UART's receive buffer.

BAUD RATES

The baud rate is the number of bits per second transferred over the serial link. To convert this to a number of bytes transferred per second, the baud rate is divided by the number of bits in the serial frame. One of the most common frames used include 8 bits of data, no parity bit, and one stop bit. Including the start bit, this adds up to 10bits. This means 9600 baud can send 960 bytes per second. To put this in more meaningful terms, it will take over 2 seconds to fill a standard 2000 character text screen at 9600 baud.

The UART has a built-in baud rate generator. The baud rate is set to match an attached device. Baud rates divisors are based on the crystal frequency. This crystal is chosen to ensure that a programmed divisor value will result in the same baud rate on any system. To determine a baud rate divisor for any baud rate, divide 115200 by the desired baud rate.

For system-to-system transfers, the baud rates can be set to very high rates, usually limited only by the CPU speed of the slower of the two systems. For a maximum rate, 115200 baud, the CPU must be able to read and store a byte every 87 microseconds.

INTERRUPT CONTROL

The serial port can be configured to operate on an interrupt (rather than polled) basis. A serial port interrupt service handler gets control whenever errors occur, when a new byte has been received, when the UART is ready to transmit a new byte, or when the modem control line status changes. These four types of interrupt action can be selectively enabled and disabled by writing to the Interrupt Enable Register.

When any of the four types of interrupts occur, the adapter card triggers an interrupt request, typically IRQ 3 OR 4. The software interrupt handler is called, typically interrupts 0Bh or 0Ch for IRQS 3 or 4. The interrupt handler must be supplied by the application using that serial port with interrupt control. **The BIOS does not provide any interrupt handling services for the serial ports.**

Once an interrupt occurs, the handler must read the serial port's Interrupt Identification Register to see the specific cause of the interrupt. When multiple interrupts occur before reading the Interrupt Identification Register, the highest priority function will be presented first. Note that within each transmitter frame ,all six channels may be varied simultaneously.

Original designs used a National Semiconductor 8250 chip. Later designs switched to a CMOS version, the 16450, that is functionally identical to the 8250. Some newer designs use a 16550 or other variant. These newer designs offer the same features as the 16450, but add additional buffering to reduce CPU overhead.

FIFO MODE

The 16550 series of UARTs operate like the common 8250/16540 UART, but provides an optional “FIFO mode”. Most systems do not use the 16550 and cannot benefit from the FIFO mode of operation.

A few other systems use another alternative UART, the 82510. This Intel chip also provides 8250 compatibility and FIFO options. It is not as common as the 16550, but does show up in some systems. The BIOS interrupts service routines will not activate FIFO mode if available. **The slow BIOS routines are avoided if high performance is required.**

With FIFO mode active, the UART will buffer up to sixteen bytes of receive data, and up to sixteen bytes of transmit data within the UART. This extra buffering allows the CPU additional time to process higher priority interrupts and other functions. Without the buffering, the data received must be read before the next byte arrives. If the CPU remains busy for an extended period of time while handling a higher priority task, and two or more bytes are received during this time, an overrun occurs. The overrun error indicates bytes were lost since the CPU did not read the data from the UART while additional bytes arrived.

Once it is determined FIFO mode is available, an alternate set of read and write routines could be substituted for the non-FIFO mode transmit and receive routines. These replacement routines can transfer multiple bytes of data at one time with the UART.

Like normal mode UART operation, the FIFO mode can be configured for interrupt or polled operation. In FIFO mode polled operation, the receive and transmit routines check the Line Status Register in the same way as the non-FIFO mode polled operation. The Line Status Register bit 0 is set when one or more bytes are in the receive FIFO. Bit 5 indicates when one or more bytes are in the transmit FIFO. Both FIFOs are used transparently

by software. This allows activation of FIFO mode while still using the BIOS serial port handler.

For FIFO interrupt operation, a trigger level is set in the FIFO Control Register. The trigger level is the number of bytes needed in the receive FIFO before a received data available interrupt is generated. The UART can be programmed to interrupt the CPU when the receive FIFO has 1, 4, 8 or 14 bytes. To keep the interrupt rate low, a higher trigger level is used. A lower trigger level is used if the CPU does not respond quickly to the interrupt. Slow CPU response can occur if higher priority interrupts are allowed or if the system is slow due to CPU clock speed or CPU choice.

In FIFO interrupt mode, the UART will also issue a receive timeout interrupt if all the three of the following conditions occur:

- The receive FIFO has one or more bytes.
- A byte has not been received over the link for the timeout period.
- The CPU has not read any bytes from the CPU in the timeout period.

The timeout period is the amount of time equivalent to receiving 4 frames in a row when no action occurs. This means the timeout is directly proportional to the baud rate. With a 10-bit frame at 9600 baud, the timeout period would be 4.2ms. For other frames and baud rates the timeout period is calculated as:

$$TimeOutPeriod = \frac{Bit\ sin\ frame}{BaudRate} * 4\ frames * 1000ms$$

The timeout duration counter is reset when a new byte is received or when the CPU reads a byte from the receive FIFO. If a timeout interrupt does not occur, it is reset when the CPU reads a byte from the receive FIFO. In interrupt mode, the transmit FIFO will generate an interrupt when the

FIFO becomes empty. The interrupt service routine can then transfer up to 16 bytes into the transmit FIFO. In interrupt mode, the transmit FIFO will generate an interrupt when the FIFO becomes empty. The interrupt service routine can then transfer up to 16 bytes into the transmit FIFO.

KEYBOARD OPERATION

The Keyboard unit contains a dedicated microprocessor that performs a variety of jobs, all of which help cut down on system overhead. The main duty of the keyboard microprocessor is to watch the keys and report to the main computer whenever a key is pressed or released. If any key is pressed continuously, the keyboard microprocessor sends out a repeat action at specific intervals.

The keyboard controller has limited diagnostic and error-checking capabilities and has a buffer that can store key actions in the rare instances that the main computer is temporarily unable to accept them.

IBM KEYBOARD

The keyboard uses a capacitive technology with a microcomputer (Intel 8048) performing the keyboard scan function.

The keyboard interface is defined so that system software has maximum flexibility in defining certain keyboard operations. This is accomplished by having the keyboard return scan codes rather than ASCII codes. In addition, all keys are typematic and generate both a make and a break scan code. Break codes are formed by adding hex 80 to make codes. The keyboard I/O driver can define keyboard keys as shift keys or typematic, as required by the application.

The microcomputer (Intel 8048) in the keyboard performs several functions, including a power-on self test when requested by the system unit. This test checks the microcomputer ROM, tests memory, and checks for stuck keys. Additional functions are: keyboard scanning, buffering of up to 20 key scan codes, maintaining bi-directional serial communications with system unit, and executing the hand-shake protocol required by each scan-code transfer.

After the processing of codes from remote control and identification of the key pressed, the key has stuffed into BIOS buffer as proper key scan codes. For this we need to have clear understanding on the underlying desktop keyboard.

The keyboard unit contains a dedicated microprocessor that performs a variety of jobs, all of which help cut down on system overhead. The main duty of the keyboard microprocessor is to detect the keys and report to the main computer whenever a key is pressed or released.. If any key is pressed continuously, the keyboard microprocessor sends out a repeat action at specific intervals. The keyboard microprocessor controller also has limited diagnostic and error-checking capabilities and has a buffer that can store key actions in the rare instance that the main computer is temporarily unable to accept them.

The PC/AT and PS/2s have sophisticated keyboard control circuitry that can perform several functions. These features include programmable typematic control, programmable scancode sets ,and improved hardware for error detection.

KEYSTROKES AND SCAN CODES

Each time when the keys on the keyboard, the keyboard circuits transmit a sequence of one or more 8-bit numbers through the connector cable to the computer. This sequence, called a scan code, uniquely identifies the key that has been pressed. The keyboard produces different scan codes, depending on whether the key was pressed or released. The keyboard control in PC/AT can also assign any of three different sets of scancode values to the keys on the 101/102 key layouts.

The 101/102-key keyboard also assigns special scan codes to certain keystroke combinations. The Alt-Sys-Req key is intended to be the same as

the Sys Req on the 84-key layout, so the 101/102-key keyboard transmits the same scan code(54h). Similar examples are given below.

101/102-key Keyboard Keystroke Combination	Scan Code transmitted
Alt-Sys-Req	54h
Print Screen	E0h, 2Ah
Ctrl-Break	E0h, 46h, E0h, C6h

Any program that processes keyboard scan codes must be aware of which machine it's running on and which keyboard is in use. Fortunately, few programs need to respond directly to keyboard scan codes --- the ROM BIOS keyboard service routines translate scan codes into meaningful information that a program can use. The following sections describe this translation processes more fully.

COMMUNICATING WITH THE ROM BIOS

The keyboard-controller circuitry on the computer's system board monitors the keyboard for input. The keyboard controller generates interrupt 09h each time it receives a byte of data from the keyboard. The ROM BIOS contains an interrupt 09h handler that reads the byte from the keyboard controller and processes it. (I/O port contains the keyboard data byte.) The interrupt 09h handler translates scan codes into 2-bytes values that are generally more useful to a program than the original scan codes.

The low-order byte of each 2-byte keyboard value contains the ASCII value corresponding to each key pressed. The high-order byte usually contains the corresponding keyboard scan code.

Special keys, such as the function keys and the numeric key-pad keys, have a 0 in the low-order byte, with the keyboard scan code in the high-order byte.

The ROM BIOS routines place the translated byte-pairs in a queue, which is kept in low memory in location 0040:001Eh. The byte-pairs are stored there until they are requested by a program, such as DOS or interpreted BASIC, that expects to read keyboard input.

TRANSLATING THE SCAN CODES

The scan-code translation job is moderately complicated because the IBM keyboard recognizes two types of keys that change the meaning of a keystroke: *shift keys* and *toggle keys*.

THE SHIFT KEYS

Three keys—Ctrl, Shift, Alt—are known as *shift* keys. They change the shift state, and thereby meaning, of whatever key they are used with. For example, when you press Shift-C, you get a capital C; when you press Ctrl-C, you generate the “break” character. The ROM BIOS recognises that all subsequent key actions will be influenced by that shift state as long as a shift key is pressed.

THE TOGGLE KEYS

In addition to the shift keys, two toggle keys also affect the keyboard’s shift state: the Caps Lock key and the NUM Lock key. When activated, Caps Lock reverses the shift state of the alphabet keys; it doesn’t affect the other keys. When activated, the NUM Lock key disables cursor-control functions on the numeric keypad. Toggle keys are activated with a single keystroke and remain active until released by a second keystroke.

The shift-key and toggle-key status information is kept by the ROM BIOS in a low-memory location (0040:0017h), where you can use or change it. When you press a shift key or a toggle key, the ROM BIOS sets a specific bit in one of these two bytes. When the ROM BIOS receives the release scan code of a shift key, it switches the status bit back to its original shift state.

Whenever the ROM BIOS receives a scan code for an ordinary keystroke, such as the letter z or a right arrow key, it first checks the shift state and then translates the key into the appropriate 2-byte code.

INTERRUPT HANDLERS

A program preparing to handle interrupts must do the following:

1. Disable interrupts, if they were previously enabled, to prevent them from occurring while interrupts are being modified.
2. Initialise the vector for the interrupt of interest to point to the program's interrupt handler.
3. Ensure that, if interrupts were previously disabled, all other vectors point to some valid handler routine.
4. Enable interrupts again.

The interrupt handler itself must follow some rigid sequence of steps:

1. Save the system context (registers, flags, and anything else the handler will modify)
2. Block any interrupts that might cause interference if they were allowed to occur during this handler's processing.
3. Enable any interrupts that should still be allowed to occur during this handler's processing.
4. Determine the cause of the interrupt.
5. Take the appropriate action for the interrupt: receive and store data from the serial port, set a flag to indicate the completion of a disk-sector transfer, and so forth.
6. Restore the system context.
7. Re-enable any interrupt levels that were blocked during the handler's execution.
8. Resume execution of the interrupted process.

SERVICING AN INTERRUPT

When the CPU senses an interrupt, it pushes the program status word (which defines the various CPU flags), the code segment (CS) register, and the instruction (IP) onto the machine stack and disables the interrupt system. It then uses the 8-bit number that was jammed onto the system bus

by the interrupting device to fetch the address of the handler from the vector table and resumes execution at that address.

Usually the handler immediately re-enables the interrupt system, saves any registers it is going to use, and then processes the interrupt as quickly as possible. Some external devices also require a special acknowledgement signal so that they will know the interrupt has been recognised. If the interrupt was funnelled through an 8259A PIC, the handler must send a special code called end of interrupt (EOI) to the PIC through its control port to tell it when interrupt processing is completed. Finally, the handler executes the special IRET instruction that restores the original state of the CPU flags, the CS register, and the instruction pointer.

Whether an interrupt was triggered by an external device or forced by software execution of an INT instruction, there is no discernible difference in the system state at the time the interrupt handler receives control. This fact is convenient when you are writing and testing external interrupt handlers because you can debug them to a large extent simply by invoking them with software drivers.

THE INTERRUPT VECTOR TABLE

The bottom 1024 bytes of system memory are called the interrupt vector table. Each 4-byte position in the table corresponds to an interrupt type (0 through 0FFH) and contains the segment and offset of the interrupt handler for that level. Interrupts 0 through 1Fh are used for internal hardware interrupts; MS-DOS uses Interrupts 20H through 3FH; all other interrupts are available for use by external hardware devices or system drivers and application software.

When an 8259A PIC or other device interrupts the CPU by means of the INTR pin, it must also place the interrupt type as an 8-bit number (0 through 0FFH) on the system bus, where the CPU can find it. The CPU

then multiplies this number by 4 to find the memory address of the interrupt vector to be used.

INTERRUPT HANDLERS AND MS-DOS

The introduction of an interrupt handler into your program brings with it considerable hardware dependence. Hardware dependencies in MS-DOS applications should be avoided whenever possible, to ensure the portability of the program.

There are valid reasons for writing our own interrupt handler for use under MS-DOS

- ✓ To supersede the MS-DOS default handler for an internal hardware interrupt (such as divide-by-zero)
- ✓ To supersede the MS-DOS default handler for a defined system exception, such as the critical handler or Ctrl-C handler.
- ✓ To chain our own interrupt handler onto the default system handler.
- ✓ To service interrupts not supported by the default MS-DOS device drivers (such as the Serial Communications port)
- ✓ To provide a path of communication between a program that terminates and stays resident and other application software

MS-DOS provides the following facilities to enable you to install well-behaved interrupt handlers in a manner that does not interfere with operating-system functions or other interrupt handlers.

Function	Action
Int 21H Function 25H	Set Interrupt Vector
Int 21H Function 35H	Get Interrupt Vector
Int 21H Function 31H	Terminate and Stay resident

These functions allow you to examine or modify the contents of the system interrupt-vector table and to reserve memory for the use of a handler without running afoul of other processes in the system or causing memory use conflicts.

Handlers for external hardware interrupts under MS-DOS must operate under some fairly severe restrictions.

- Because the current versions of MS-DOS are not re-entrant, a hardware interrupt handler should never call the MS-DOS functions during the actual interrupt processing.
- The handler must re-enable interrupts as soon as it gets control, to avoid crippling other devices or destroying the accuracy of the system clock.
- A program should access the 8259A PIC with great care. The program should not access the PIC unless that program is known to be the only process in the system concerned with that particular interrupt level. And it is vital that the handler issue an end-of-interrupt code to the 8259A PIC before performing the IRET; otherwise, the processing of further interrupts for that priority level or lower priority levels will be blocked.

The following are a few rules that should be followed when writing an interrupt driver

- ✓ Use 21H Function 25H to modify the interrupt vector. Do not write directly into the system Interrupt Vector Table.
- ✓ Chain back to the previous interrupt handler after performing processing of an interrupt.
- ✓ If the program is going to stay resident, fetch and save the current contents of the interrupt vector before modifying it and then restore the original contents when the program exits.
- ✓ If the program is going to stay resident, use terminate-and-stay resident functions to reserve the required amount of memory.

If hardware interrupts are processed, keep the time that interrupts are disabled and the total length of the service routine to an absolute minimum.

MEMORY-MAPPED DISPLAY TECHNIQUES

Display performance is best when an application program takes over complete control of the video adapter and the refresh buffer. Because the display is memory-mapped, **the speed at which characters can be put on the screen is limited only by the CPU's ability** to copy bytes from one location in memory to another. Since our routine is a real time sampling code we need to do our screen output at that maximum possible speed.

TEXT MODE:

Direct programming of the IBM PC-compatible video adapters in their text display modes is straightforward. The character set is the same for all, and the cursor home position is defined to be the upper left corner of the screen. Each character-display position is allotted 2 bytes in the regen buffer. The first byte contains the ASCII code of the character, which is translated by a special hardware character generator into a dot-matrix pattern for the screen.

The second byte is the attribute byte. Several bit fields in this byte control such features as blinking, intensity, and reverse video, depending on the adapter type and display mode.

0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	White
8	Gray

9	Light Blue
10	Light Green
11	Light Cyan
12	Light Red
13	Light Magenta
14	Yellow
15	Intense White

The memory offset of any character on the display as the line number (y co-ordinate) times 80 characters per line times 2 bytes per character, plus the column number (x co-ordinate) times 2 bytes.

Offset = ((y * 50H + x) * 2) + (page * 1000H) in 80 by 25 text modes
= ((y * 50H + x) * 2) + (page * 0800H) in 40 by 25 text modes

The following code stores the character and attribute into the MDA's video refresh buffer at the proper location:

```

push ax          ; save char and attribute
mov  ax,160
mul  cx          ; DX:AX = Y * 160
shl  bx, 1       ; multiply x by 2
add  bx, ax      ; BX = ( Y * 160 ) + ( X * 2 )
mov  ax, 0b000h  ; ES = segment of monochrome
mov  es, ax      ; adapter refresh buffer
pop  ax          ; restore char and attribute
mov  es:[bx], ax ; write them to the video buffer

```

Programs that write characters directly to the CGA region buffer in text modes must deal with an additional complicating factor -- they must examine the video controller's status port and access the refresh buffer only

during the horizontal retrace or vertical retrace intervals. (A retrace interval is the period when the electron beam that illuminates the screen phosphors is being repositioned to the start of a new scan line.) Otherwise, the contention for memory between the CPU and the video controller is manifest as unsightly "snow" on the display. In EGA, MCGA, and VGA, we can ignore the retrace intervals, since snow is not a problem with these controllers.

USING THE SYSTEM TIMER TICK

In the PC/XT/AT family, the timer chip has three output channels, each with a particular dedicated function:

⊕ Channel 0 is the system clock-timer. When the computer is cold booted, the ROM BIOS programs the timer oscillate with a frequency of about 18.2 ticks per second. This signal is tied to the computer's interrupt controller in such a way that interrupt 08H is generated each time the clock ticks.

⊕ Channel 1 is always dedicated to producing the RAM refresh timing signal; it's not intended for use in software applications.

⊕ Channel 2 is used to control the computer's speaker; The frequency of the timer's channel 2 signal determines the frequency of the sound emitter by the loudspeaker.

In all PCs and PS/2s, the interrupt oscillator to the system timer circuit has a frequency of 1.19318 MHz. On each cycle, the timer chip decrements the values in a set of internal 16-bit counters, one for each of the timer's output channels. When the value in a counter reaches 0, the chip generates a single output pulse on the corresponding channel, resets the count, and starts counting down again.

When the ROM BIOS initialises the system timer, it stores a countdown value of 0 in the count register for channel 0. this means that the timer chip decrements the counter 2¹⁶ times between output pulses on channel 0, so output pulses occur 1,193,180/65,536, or about 18.2 times per second.

The output from timer channel 0 is used as the signal on interrupt request level 0 (IRQ0), so interrupt 08H occurs whenever channel 0 of the system timer counts down to 0 - that is, 18.2 times per second.

The ROM BIOS contains an interrupt handler for interrupt 08H that increments a running count of clock ticks at 0040:006CH in the BIOS data area. The ROM BIOS interrupt 08H handler also issues software interrupt 1CH, which is intended for use in programs that want to be notified when a system timer tick occurs. A program can detect when each timer tick occurs simply by pointing the interrupt 1CH vector at 0000:0070H to its own interrupt handler. If you use an interrupt 1CH handler in a program, however, be aware that the ROM BIOS interrupt 08H handler does not allow subsequent clock-tick interrupts on IRQ0 to occur until your interrupt 1CH handler returns.

The system timer tick and its interrupt are useful in programs that must perform a simple task at a regular interval regardless of what else is going on in the computer. The timer-tick interrupt has the highest priority of any of the hardware interrupts (except the non maskable interrupt), so the code in the corresponding interrupt 08H and 1CH handlers takes precedence over all other system software.

SOFTWARE: HAND-HELD REMOTE KEYPAD

The Sanyo Code contains 16 bits of data. This when sampled amounts to 32 bits of data. However, eight of these bits are lost in the stop and start bits. Therefore, we have 24 bits or 3 bytes of data.

The total number of code combinations is 18. The received codes for all the 18 keys are given below:

Key Pressed	Code Received (in Hex)
1	D5, DD, D7
2	75, 77, 5D
3	75, DD, 75
4	D7, 57, 75
5	5D, 77, 77
6	5D, 5D, DD
7	DD, DD, 75
8	DD, D7, D5
9	D7, 5D, 5D
0	55, 77, D7
Select	55, 77, 77
Enter	57, 75, 77
Vol. Up	77, 57, 75
Vol. Down	77, 5D, 5D
Ch. Up	77, 5D, 77
Ch. Down	77, 5D, 5D
Mute	57, D7, 77
A/B	55, 77, 77
P/C	55, 77, 77
Timer	75, DD, DD

Key Pressed	Code Received (in Hex)
Delete	D7, 5D, 5D
Prog.	77, 57, D7
Code	55, 77, D7
Power	57, 75, DD

When a keypress is detected, the received pattern is compared with the patterns in the already saved data set, and the corresponding key is stuffed into the keyboard buffer. If all the three bytes are compared, in the worst case, it would require about 70 microseconds to complete the ISR. Other interrupts should not be disabled for this amount of time.

Hence we have used the redundancy in the 24 bits and reduced it to 8 bits, which would reduce the time consumed to a great extent.

An interesting question is why should not the data be read at the same speed as the input signal. To avoid synchronisation problems and lost of data.

The compression algorithm is explained below:

Message byte 1: M 1 M 1 M 1 M 1
Message byte 2: M 1 M 1 M 1 M 1
Message byte 3: M 1 M 1 M 1 M 1
(‘M’-message bits)

All even bits in Message bytes 1 and 2 are reset.

The 2nd Message byte is rotated through right by 1 bit. Message bytes 1 and 2 are ORed. Then the third byte is XORed with the resultant bit.

This algorithm is tested for all code combinations of Sanyo Code, and it preserves the uniqueness of the code.

The following comparison of different techniques to access the serial port, clearly depicts why we decided in favour of direct access method (to control the serial port).

MS-DOS

- Poor support for Serial port
- Slow and primitive implementation of services

ROM BIOS

- Only Polling method supported.
- No Interrupt method, to inform us of available data.
- Not Practical in Real time.
- Baud Rate support is limited (like 2000Hz is not available)

DIRECT ACCESS

Advantages

- ☑ Full Control of ports
- ☑ All Baud rates possible are supported
- ☑ Real time
- ☑ Interrupt method supported which reduces CPU overload

DownSide

- ⇒ Full responsibility on the stability of the service and the system is on the programmer
- ⇒ Very tedious to implement

DEFINITIONS FOR SERIAL PORT INITIALIASING MODULE

(The port assumed here is COM1)

```
#define LSB_PORT          0x3f8
#define MSB_PORT          0x3f9
#define REC_PORT          0x3f8
#define INT_EN_PORT       0x3f9
#define INT_ID_PORT       0x3fa
#define LIN_CON_PORT      0x3fb
#define MOD_CON_PORT      0x3fc
#define LIN_STA_PORT      0x3fd
#define MOD_STA_PORT      0x3fe
#define SCR_PAD_PORT      0x3ff
#define PIC_MASK_PORT     0x21
#define PIC_INT_ENA       0xef
#define PIC_EOI_PORT      0x20
#define PIC_EOI_REG       0x20

#define LO_DIV            0x3A
#define HI_DIV            0x00
#define LIMIT             3

#define COM_INT_VEC       0x0c      /* 0000 1100 */
#define LIN_REG            0x03      /* 0000 0011 */
#define IE_REG            0x01      /* 0000 0001 */
#define ID_REG            0x06      /* 0000 0110 */

#define MOD_CON_REG       0x0B      /* OUT2 High*/
```

INTERRUPT SERVICE ROUTINE

```

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
* "status" is global variable for storing line status      *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * */
void interrupt New_Isr() {
/* Read the Line Status Port */

status = inp (LIN_STA_PORT);

/* Check for Data Ready signal and read it */
if (status & 0x01 == 1) Identify(inp(REC_PORT));

outp(PIC_EOI_PORT, PIC_EOI_REG); /* Send EOI to 8259 */
}

```

TIMER MODULE FOR CHECKING INCOMPLETE CODES

```

/* ***** */

* "c" is a global variable for count of received codes *

*****/

void interrupt New_Timer() {
    /* check for incomplete pattern and reset */
    if ((c==2) || (c==3)) c=1;
    (*Old_Timer)();
}

```

```

/* **** */
* "temp" is a static global variable *
* "Old_Isr" & "Old_Timer" are old isr address *
* "New_Isr" & "New_Timer" are new isr address *
*****/

void Set_Serial(){
    outp (LIN_CON_PORT,LIN_REG); /* Read the Line Register */
        /* Make Bit 7 High of Line Register*/
    outp (LIN_CON_PORT,LIN_REG | 0x80);
    outp (LSB_PORT,LO_DIV); /* Set Baud Rate for Low Divisor */
    outp (MSB_PORT,HI_DIV); /* same as above for Hi Divisor */
    outp (LIN_CON_PORT,LIN_REG); /* Reset Line Reg. to 7th Bit */
    outp(LIN_STA_PORT,0x00); /* Init the registers */
    outp(REC_PORT,0x00);
    outp (INT_EN_PORT, IE_REG); /* Set the Int. Register of 8250 */
    outp (INT_ID_PORT, ID_REG); /* Set the Int ID Reg. of 8250 */
    Old_Isr = getvect(COM_INT_VEC); /* Get the old ISR */
    setvect (COM_INT_VEC,New_Isr); /* Set our ISR for 0C/0B */
    Old_Timer = getvect(0x08) ; /* Get the old timer */
    setvect (0x08, New_Timer) ; /* Set our own TIMER ISR */
    disable(); /* disable interrupts */
        /* Get value from PIC Mask*/
    temp=(unsigned char) inp(PIC_MASK_PORT) ;
        /* Set COM1 Enable by 8259 Mask*/
    outp(PIC_MASK_PORT, temp & PIC_INT_ENA);
        /*Set the Modem Control
    outp(MOD_CON_PORT, MOD_CON_REG);
        Register of 8250 */
    enable(); /* enable interrupts */
    return;
}

```

```
void Identify (unsigned char Value) {
/* * * * * * * * * * * * * * * * * * * * * * * * * * */
    * "Buffer" is a memory array with the input key pattern *
    * "Tail" is the total no. of key codes *
    * "Codes" is a memory array with all key codes and ID's *
    * "KeyChar" is a memory array of key board scan codes *
    * * * * * * * * * * * * * * * * * * * * * * * */

    unsigned char i=0;
    unsigned char CodeValue;
    static int z=0;

    if (Value==255) return; /* If NULL discard and return
                                NO ACTION TAKEN*/

    Value |= 0x55;      /* ERROR CORRECTION */
    *(Buffer+c) = Value;
    if (c == 3 {   c = 1;
        CodeValue=(*(Buffer+1) & 0xAA) ^ (*(Buffer+2) & 0xAA) >> 1) ;
        CodeValue ^= (*(Buffer+3)) << 1) ;
        for (i=0;i<Tail;i+=2)
            if (CodeValue == *(Codes+i)) {
                /* the key pattern is stuffed into BIOS area */
                stuffkey(*(KeyChar + *(Codes+i+1)), 0x23);
                return ;
            }
        }
    else c++;      /* see the next item in the pattern */
    return;
}
```

KEYBOARD KEY STUFF MODULE

```
void stuffkey (char cAscii, char cScan) {
    unsigned far *buffer_tail = (unsigned far *) 0x0040001C;
    unsigned far *buffer_end = (unsigned far *) 0x00400082;
    unsigned far *buffer_head    = (unsigned far *) 0x0040001A;
    unsigned far *buffer_start    = (unsigned far *) 0x00400080;

    char far *ascii              = (char far *) 0x00400000;
    char far *scancode           = (char far *) 0x00400000;

    ascii[*buffer_tail]         = cAscii;
    scancode[*buffer_tail+1]     = cScan;

    *buffer_tail += 2;
    if (*buffer_tail >= *buffer_end) *buffer_tail = *buffer_start;
}
```

The keys in the remote that we mapped to desktop keyboard are as follows:

REMOTE KEYS	DESKTOP KEYBOARD KEYS
1,2,3,4,5,6,7,8,9,0	1,2,3,4,5,6,7,8,9,0
SELECT	SPACE BAR
ENTER	ENTER/RETURN
VOLUME UP	UP
VOLUME DOWN	DOWN
MUTE	BACKSPACE
TIMER	TAB
PROG	F1
POWER	ESC

PROBLEMS FACED

The initial design for the KeyPad was to have a parallel RC-5 decoder after the sensor. This approach using an IC like SAA3049 initiated us to work in the lines of programming the Parallel port as input for six data bits. Due to the non-availability of SAA3049 or any equivalent IC, within the specified time limit, the parallel approach couldn't be accomplished. This initiated us to look into the Serial Port, which could be programmed for any desired frequency upto 9600Hz.

Even though BIOS extends support for fixed baud rates like 1200Hz, 2400Hz, etc., there was no support for our frequency requirements. This made us resort to direct programming of the Serial Port.

Spurious noise signals emitted by other sources such as incandescent lamps, fluorescent lamps, were read by the system initially. These were later filtered out. Improper keypresses generate incomplete signals which are neglected by the system at a later stage.

APPLICATIONS

A few of the many possible applications for "Hand Held Remote KeyPad" are outlined below:

- ✓ Numeric data entries can be effectively done without being close to computers thus facilitating easier gathering of data.
- ✓ Inside Nuclear Research Stations where dangerous areas are enclosed by transparent enclosures infra red can be used since electromagnetic signals cannot be used.
- ✓ Super Markets can use the system in place of more expensive bar code readers. Even where bar code readers are a must, the keypad can be attached to it.
- ✓ Can be used in Restaurants for ordering the necessary items for required quantity.
- ✓ In dedicated computer installations where the use of full fledged keyboard is not possible/necessary, "KeyPad" can be used since it is compact and provides adequate number of keys for these applications. A classic example of this is a computer based Car Wheel Aligner system.
- ✓ In the Control of computer based presentations through Video Projectors, the slides can be changed up and down even if the speaker is close to the audience and far from the computer.
- ✓ Doctors can use it for controlling patient monitoring equipments inside Operation Theatres.
- ✓ On the lighter side, the KeyPad can be used for playing realistic computer games from a convenient distance.

Bibliography.

1. IBM PC/XT Technical Reference Manual
 2. Motorola Linear IC manual
 3. RS “Infra-Red Devices”, Data Sheet no.F18528
 4. Elektor India Magazine
 - 4.1. June 1993
 - 4.2. November 1994
 - 4.3. April 1995
 5. OptoElectronics Circuits Manual - *R.M.Martson*
 6. Advanced MS-DOS Programming - Ray Duncan
 7. The IBM PC & PS/2 - *Peter Norton*
 8. The Personal Computer the Inside and Out - *Sargent & Shoemaker*
 9. Undocumented PC - *Frank Van Gilluwe*
 10. Undocumented DOS - *Andrew Schulman*
 11. The Waite Group’s Turbo Assembler Bible - *Gary Syck*
-

APPENDIX - A

IR SENSOR. TECHNICAL SPECIFICATIONS

The IS1U60 is an IR remote control receiver. This device is capable of a modulated 38Khz signal and converting it into a logic pulse train output. internal EMI shield eliminates for metal housing. It is in a compact 3 pin package.

ABSOLUTE MAXIMUM RATINGS:

<u>Parameter</u>	<u>Symbol</u>	<u>Rating</u>	<u>Unit</u>
Supply voltage	V _{CC}	0~6.0	V
Operating temperature	T _{opr}	-10~+60	°C
Soldering temperature	T _{sol}	260	°C

ELECTRO CHARACTERISTICS (TA=25°C, V_{CC}=5V)

<u>Parameter</u>	<u>Symbol</u>	<u>Min</u>	<u>Typ</u>	<u>Max</u>	<u>Unit</u>
Operating supply vol.	V _{CC}	4.7	-	5.3	V
Dissipation Current	I _{CC}	-	2.8	4.5	mA
High level o/p vol.	V _{OH}	V _{CC} -	-	-	V
Low level o/p vol.	V _{OL}	-	0.45	0.6	V
High level pulse width	T ₁	400	-	800	μs
Low level pulse width	T ₂	400	-	800	μs
Peak Sensitivity	-	-	1000	-	nm
B.P.F frequency range	f ₀	36	38	41	kHz
Reception distance	L	5.0			m

APPENDIX-B

PHILIPS REMOTE TRANSMITTER SPECIFICATIONS®

ELECTRICAL:

Transmission/Coding system	:	Philips RC-5 (enlarged)
Wavelength	:	940nm
Transmitting power	:	100mW/sr or 200mW/sr
Operating distance	:	20m with a receiver sensitivity of 0.25 mW/sr
Modulation frequency	:	36Khz
Batteries	:	AAA
Battery life	:	30h on continuous operation, 1 year on normal operating conditions

MECHANICAL:

Size	:	156 x 36 x 16 (mm)
Operating temperature	:	+5°C to +45°C

The Philips RC5 coding system provides upto 2048 commands arranged in 32 addressable groups of 64 commands each. RC5 infra red transmissions are immune to sunlight, incandescent lamps and fluorescent tubes. Moreover the 36Khz carrier frequency used is immune from TV line-scan interference.

APPENDIX C

PHOTOMETRIC TERMINOLOGY

This is brief introduction to the basics of photometry. Photometry forms the underlying principles in Infra-Red devices working.

FLUX(SYMBOL F)

Any radiation, whether visible or otherwise, can be expressed by a number of flux lines about the source, the number being proportional to the intensity of that source. This luminous flux is expressed in LUMENS for visible radiation.

LUMINOUS EMITTANCE (SYMBOL L)

A source measurement parameter. It is defined as the ratio of the luminous flux emitted from a source to the area of that source, or $L = F/A_e$. Typically expressed in “PHOT” or “LUX” or “FOOT CANDLE”.

ILLUMINANCE (SYMBOL E)

This is a target or detector area measurement parameter. It is the ratio of flux lines incident on a surface to the area of that surface or $E = L/A_d$. Typical Measurement units are same of Luminous Emittance.

LUMINOUS INTENSITY (SYMBOL I)

A spatial flux density concept. It is the ratio of luminous flux of a source to the solid angle subtended by the detected area and that source. The Luminous Intensity of a source assumes that source to be point rather than an area dimension. The Luminous Intensity of a source is measured in LUMENS/STERADIAN, which is equal to one “CANDELE”.

LUMINANCE (SYMBOL B)

Sometimes called photometric brightness (although the term brightness should not be used alone as it encompasses other physiological factors such as colour, sparkle, texture, etc.) it is applied to sources of appreciable area size. Mathematically, if the area of an emitter (circular for example) has a diameter or diagonal dimension greater than 0.1 the distance to the detector, it can be considered as an area source. If less than this 10% figure, the source can be treated as point in nature. This one to ten ratio of source diameter to distance is offered as it mathematically very closely approximates results obtained when comparing an area source to its point equivalent. Luminance presents itself as an extremely useful parameter as it applies a figure of merit to:

1. Apparent or projected area of the source (A_p).
2. Amount of luminous flux contained within the projected area of the source.
3. Solid angle the projected area generates with respect to the centre of the source.

CIE CURVE

Photometric quantities are related to the corresponding radiometric quantities by the CIE standard Luminous Function which is often called the 'standard eyeball'. The eye responds to the rate at which radiant energy falls on the retina, i.e., on the radiant flux density expressed as Watts/m². The corresponding photometric quantity is Lumens/m². The standard luminosity function is then, a plot of Lumens/Watt as a function of wavelength.

DISCRETE INFRA-RED PAIRS

The miniature versions comprise a solution-grown pn junction. GaAs infra-red emitting diode spectrally matched to a silicon passivated NPN

planar phototransistor which has exceptionally stable characteristics and high illumination sensitivity.

The high power pair consists of high radiant intensity PN GaAs infra-red emitting diode spectrally matched to a high photo sensitivity silicon PIN photo diode .The high power pair is recommended for use with the remote control IC's.

APPENDIX-D

HOW THE 8086 USES I/O PORTS

The 8086-family microprocessors communicate with and control many parts of the computer through the use of input and output (I/O) ports. The I/O ports are doorways through which information passes as it travels to or from an I/O device, such as a keyboard or a printer.

Most of the support chips are accessed through I/O ports; in fact, each chip may use several port addresses for different purposes. Each port is identified by a 16-bit port number, which can range from 00H through FFFFH(65,535). The CPU identifies a particular port by the port's number.

As it does when accessing memory, the CPU uses the data and address buses as conduits for communication with the ports. To access a port, the CPU first send a signal on the system bus to notify all I/O devices that the address on the bus is that of a port. The CPU then sends the port address. The device with the matching port address responds.

The port number addresses a memory location that is associated with an I/O device but is not part of main memory. In other words, an I/O port number is not the same as a memory address. For example, I/O port 3D8H has nothing to do with memory address 003D8H. To access an I/O port, you use instructions IN and OUT, which are reserved for I/O port access.

SOME IMPORTANT I/O PORT NUMBERS:

System Timer	40H-5FH
Keyboard controller	60H-6FH
CGA	3D0H-3DFH
Serial Communications 1	3F8H-3FFH

APPENDIX E

INTERRUPTS AND THE 80x86 FAMILY:

Interrupts are signals that cause the computer's central processing unit to suspend what it is doing and transfer to a program called an interrupt handler. Special hardware mechanisms that are designed for maximum speed force the transfer. The interrupt handler determines the cause of the interrupt, takes the appropriate action, and then returns control to the original process that was suspended.

Interrupts are typically caused by events external to the central processor that require immediate attention, such as the following:

- ➔ Completion of an I/O operation
- ➔ Detection of a hardware failure
- ➔ "Catastrophes"

In order to service interrupts more efficiently, most modern processors support multiple interrupt types, or levels. Each type usually has a reserved location in memory, called an interrupt vector; that specifies where the interrupt-handler program for that interrupt type is located.

This design speeds up processing of an interrupt because the computer can transfer control directly to the appropriate routine; it does not need a central routine that wastes precious machine cycles determining the cause of the interrupt. The concept of interrupt types also allows interrupts to be prioritised, so that if several interrupts occur simultaneously the important one can be processed first.

CPUs that support interrupts must also have the capability to block interrupts while they are executing critical sections of code. Sometimes the CPU can block interrupt levels selectively, but more frequently the effect is global. While an interrupt is being serviced, the CPU masks all other

interrupts of the same or lower priority until the active handler has completed its execution; similarly, it can preempt the execution of a handler if a different interrupt with higher priority requires service. Some CPUs can even draw a distinction between selectively masking interrupts and simply disabling them.

The Intel 80x86 family of microprocessors support 256 levels of prioritised interrupts, which can be triggered by three types of events.

- ➡ Internal hardware interrupts
- ➡ External hardware interrupts
- ➡ Software interrupts

INTERNAL HARDWARE INTERRUPTS

Internal Hardware interrupts, sometimes called faults, are generated by certain events encountered during program execution, such as attempt to divide by zero. The assignment of such events to certain interrupt numbers is wired into the processor and is of modifiable.

Interrupt level	Vector Address	Interrupt Trigger
00 h	00h - 03 h	Divide by zero
01 h	04h - 07 h	Single step
02 h	08h - 0B h	Nonmaskable Interrupt (NMI)
03 h	0Ch - 0Fh	Break point

EXTERNAL HARDWARE INTERRUPTS

External hardware interrupts are triggered by peripheral device controllers or by coprocessors. These can be tied to either the CPU's nonmaskable interrupt (NMI) pin or its maskable interrupt (INTR) pin.

The NMI line is usually reserved for interrupts caused by such catastrophic events as a memory parity error or a power failure.

Instead of being wired directly to the CPU, the interrupts from external devices can be channelled through a device called the Intel 9259A Programmable Interrupt Controller.

SOFTWARE INTERRUPTS:

Any program can trigger software interrupts synchronously simply by executing an INT instruction. MS-DOS uses Interrupts 20H through 3FH to communicate with its modules and with application programs. The IBM PC ROM BIOS and application software use other interrupts, with either higher or lower numbers, for various purposes.

Interrupt	Usage
20 h	MS DOS terminate process
21 h	MS DOS function dispatcher
22 h	MS DOS terminate address
23 h	MS DOS Ctrl-C handler

The text for this report was prepared using Microsoft Edit. It was processed and formatted using Microsoft Word for 95 under Windows NT 3.51.

The Circuit diagrams were prepared using ORCAD v4.2.

The block diagrams and schematic diagrams were done using MS-Paintbrush and VISIO 4.0 for Windows 95.

The images were scanned using Genius Scanmate/Color SE, handheld colour scanner. The report was printed with Hewlett Packard Deskjet 520 and 660C printers.