Robin: An Experimental Protocol for Asynchronous Serial Lines

by

Hon Pan Chum

A Research Paper
Submitted to
Oregon State University

In partial fulfillment of
the requirements for the degree of
Master of Science

December 9, 1988

ACKNOWLEDGEMENT

I would like to thank my major advisor Dr. Swart for his year long guidance and encouragement. Having had two classes and a research project under him was most beneficial to me. Mr. John Sechrest give me the most help because of his excellent understanding on computer networks and operating systems. He was always there when I needed him most. I was really enjoy to discuss with him.

I would also like to thank my family for their support on my studies. A lot of encouragement came from Chinese Christian Fellowship and I thank fot that.

Table of Contents

1. Introduction

Computer networking research has been going on for almost twenty years; however, most of the protocols concern high and low speed synchronous transmission. All synchronous communication needs special transmission lines. The TCP/IP (Transmission Control Protocol / Internet Protocol) protocols, Ethernet, X.25, X windows system are very successful protocols. Less work has been done on low speed asynchronous serial communication. Asynchronous communication software can be applied to the telephone system by using a modem to transform digital signals into analog signals. Telephone (voice grade) lines are established world wide; therefore, any improvement in asynchronous protocols affects our entire society. Currently, software developers rely on improved hardware, like the Integrated Services Data Network (ISDN) [4], a high bandwidth communication channel for voice, data, image, etc. Switching to this technology is a major change for telephone companies, and won't be in common use for some years [6]. There are a number of commercial software products providing serial communication such as: xmodem, Kermit, procomm; etc. However, they are slow because they do not fully utilize the transmission line. Research seeks to improve telephone data transmission by making efficient use of limited communication capabilities by giving high priority to interactive activities while simultaneously supporting less critical requests. An experimental protocol has been designed which can transfer files in background , and remote log-in to a host machine at the same time.

1.1 The Main Problem

The major problem of using the telephone line for transmitting data is its low speed. The cutoff frequency on an ordinary telephone line is near 3000 Hz [7, pp.93]. The maximum bit rate achieved by modems now is 19200 bps [7, pp.93], but such modems are expensive and uncommon. The processing speed of a 20 MHz microprocessor is obviously much faster than input and output. Therefore, the communication bottleneck between a user's microcomputer and a

host machine is the telephone line. If we use a typical protocol for high speed transmission on a serial line, there is a big overhead cost. For example, a TCP/IP packet consists of 20 - 24 bytes of IP header, 20 - 24 bytes of TCP header which is a very inefficient way to transmit one byte of data. The internetworking ability of these protocols implies they need extra space to put source address, destination address and other control flags into the headers. However, since we are concerned with point-to-point serial communication between a microcomputer and a more powerful host machine, 40 bytes of TCP/IP overhead is not acceptable, especially on a serial line. A new protocol called SLIP (Serial Line Internet Protocol) [5] has been developed that replaces the Ethernet to become the underlaying protocol of TCP/IP in Unix systems. However, SLIP does not solve the overhead problem. Kermit [3], designed at Columbia University, is a point-to-point communication protocol offered by most of communication systems. It deals well with asynchronous serial lines but works in half duplex and cannot handle both a remote log-in session and a file transfer session simultaneously. Half duplex transmits data between machines alternately and slows down the whole communication.

1.2 The Goal of Robin

Robin is a protocol that supports multiple sessions and to rapidly send very small packets to support remote log-in from a terminal emulator through a serial line. To fully utilize an asynchronous serial line and be portable, we include a full duplex, single process, user level application in our design. It involves designing a Data Link Layer which is the second lowest level according to the ISO OSI Reference Model [10]. The application part of Robin is a file transfer capability which is a low priority background job from the user's point of view. The file transfer session can be interrupted at almost any time to allow a remote command to be issued.

2. Architecture of Robin

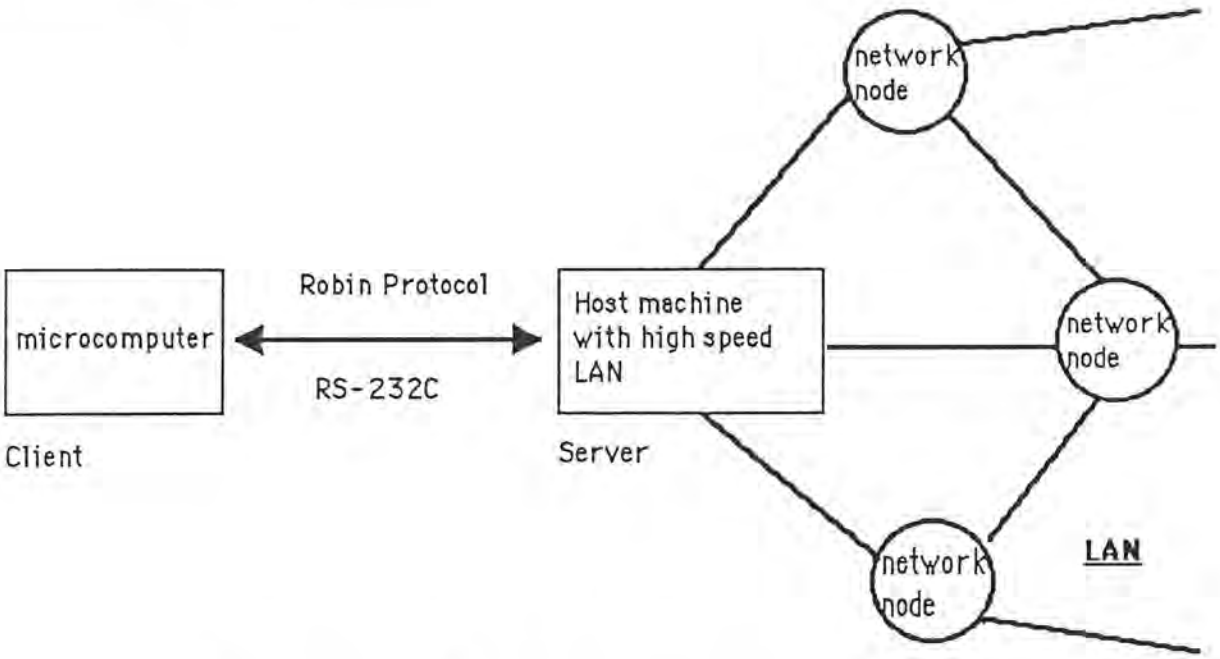

Fig. 2-1 The configuration of Robin between a microcomputer and a host machine with LAN

An overview of where the Robin protocol is used is shown in Fig. 2-1 (see Appendix A for microcomputer and host machine hardware requirements). Basically, Robin is divided into two layers: Upper Robin (UR) and Lower Robin (LR). The UR deals with multiple sessions, asynchronous read/write, file transfer, and sliding windows. The LR deals with input/output state diagrams, error checking and framing packets. Robin is based on the Client/Server Model [9] which allows client applications to request services from a server process. Both client and server can be a sender and/or receiver because of the full duplex capability of the protocol. Therefore, a client process initiates the whole communication. Fig. 2-2 is the schematic diagram of the Client/Server Model.

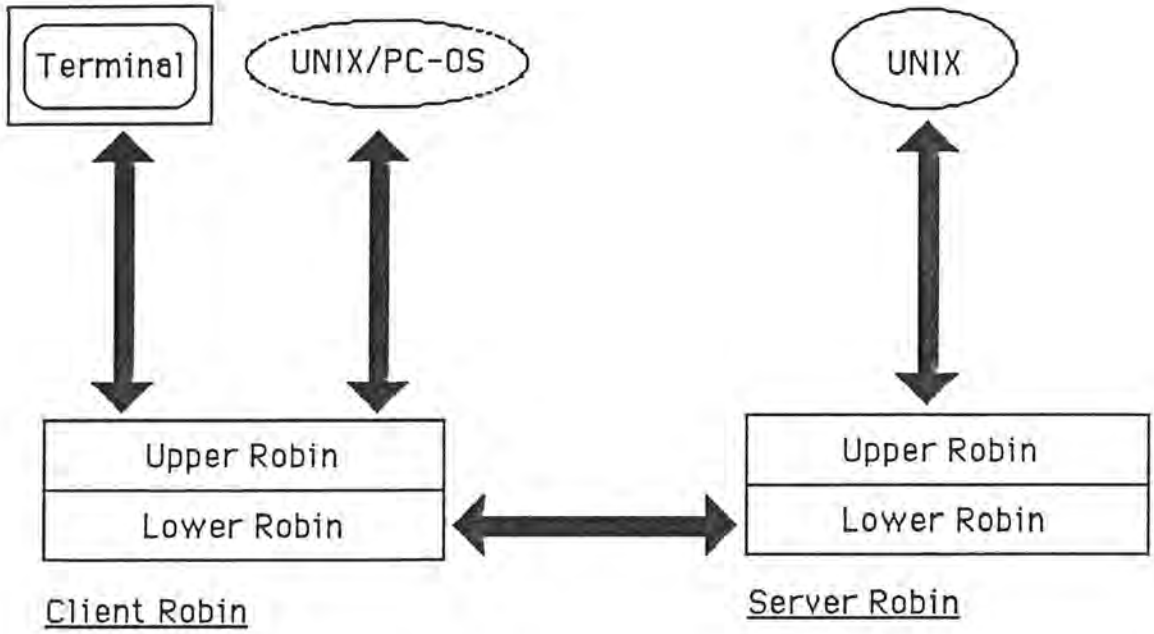

Fig. 2-2 Schematic diagram of Client/Server Model in Robin

Section 2.1 discusses formats for data, commands and packets. Input/output state diagrams which handle all incoming or outgoing data, control commands, and packets are discussed in section 2.2. Section 2.3 describes upper Robin protocol.

2.1 Data, Command and Packet Formats

Robin is a character-oriented protocol. Some characters must be attached at the beginning and end of a block of data to form a packet. To avoid mistaking the header character (start of packet) as data, escape characters are used. There are hot characters, packets and commands which are sent across the line. All of these will be discussed in the following subsections.

2.1.1 Escape Characters

Most of the existing asynchronous protocols are based on transferring packets. In Robin, packets are interruptible by data, or "hot characters" (see section 2.1.2), which need to be sent immediately and are usually short. To distinguish a hot character from a character in a packet, one escape character, called the Interrupt Escape character (INTESC, octal 024), is used in front of each hot character when it is inserted in a packet. Another escape character, called the Control Escape character (CTLESC, octal 021), is used as a header of a packet or control command.

These two escape characters have special meanings to the Robin protocol. When escape characters appear in a packet as data, the receiver would be confused; therefore, we have a special interpretation for the following characters. To allow the receiver to receive intended character patterns, the sender inserts an extra CTLESC in front of any CTLESC and INTESC inside a packet. Double CTLESC implies that CTLESC is part of data. The sequence of CTLESC INTESC implies that INTESC is part of the data. For example, suppose the data given to UR is Fig. 2-3 (a). The LR inserts escape characters into the data before sending it out as Fig. 2-3 (b). This method for achieving data transparency is known as character stuffing [7, pp.165] .

(a) . . CTLESC . . . INTESC . . . CTLESC INTESC . . .

(b) . . CTLESC CTLESC . . . CTLESC INTESC . . . CTLESC CTLESC CTLESC INTESC . . .

Fig. 2-3 Escape characters sequence on incoming data.

2.1.2 Hot Character

Hot characters are data which are sent immediately without framing or packaging. In some sense, they are "raw" data which was received and sent without being processed. A telephone line is not reliable because of the electrical noise. No packaging implies hot characters could easily have undetected errors. Therefore, a parity bit is used on each hot character for minimal error checking.

One parity bit and a seven bit ASCII character make up one hot character.

For most communication protocols, packets are not interruptible. If one wants to send a small amount of data, even one character, the sender has to wait for completion of the current packet. With Kermit, a user cannot input any commands to the remote log-in session before completion of the entire file transfer. In Robin, hot characters can be "inserted" into packets. They are distinguished by an INTESC in front of each of them. They are especially useful for remote log-ins to a host machine because shell commands issued by a user are usually short. All hot characters are received by the host machine and then sent to the shell of the host to execute. The result will also be sent back as hot characters. We assume that issuing shell command in remote log-in session is short and urgent. For example, if a user "cats" a big file to a terminal (i.e. reading from the terminal) during file transfer, that will only slow down the whole protocol.

2.1.3 Control Command

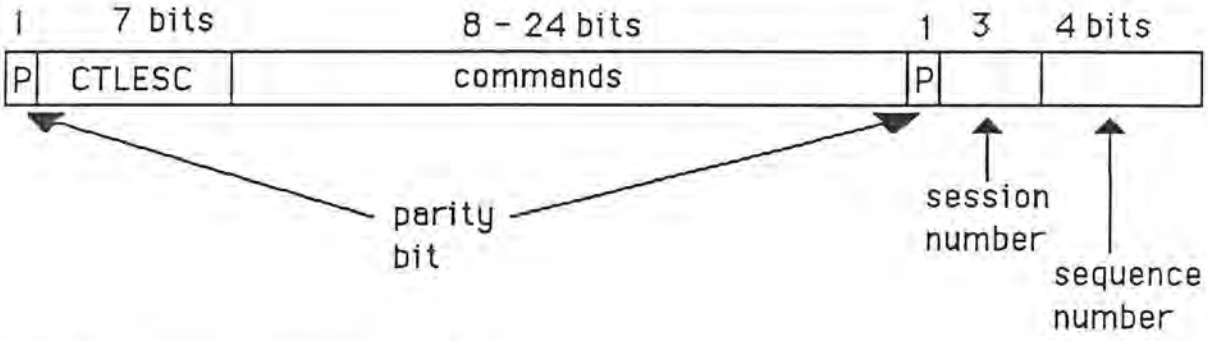

Fig. 2-4 A control command format.

A control command is a sequence of ASCII bytes, usually less than 4 bytes, which controls the operation of the line and the communication between two applications. Each byte has a parity bit. There are two types of commands, one used for verification and error checking of Robin,

including acknowledgement, negative acknowledgement, etc.; and, the second used to establish control over the line, such as open, close and reset a session. LR handles the first type of commands while UR handles the second type.

A control command begins with a CTLESC, one to four byte sequence of commands, and one byte with three bits for session number (see section 2.2.2), four bits for sequence number (see section 2.1.4) and one bit for parity (Fig. 2-4).

2.1.4 Packet

A packet is a block of data with a header and a trailer which are used for control. Bytes have no parity bit except for control bytes in the header and trailer. Packets are used only for file transfers at this stage of development. The maximum size for a packet is 1K bytes, and there can be empty packets. The size of a packet is determined by the UR. Since the data can be any kind (e.g. ASCII text, binary, graphic image), a checksum is calculated at the end of a packet to detect transmission errors. We will not deal with the problem of incompatible file types between a microcomputer and a host machine [3]. The function of a packet is to transfer a block of data safely without looking at the data itself.

Each packet begins with a header which containing a CTLESC, BOP (Beginning Of Packet) and a session number (three bits), a sequence number (four bits) and one parity bit combination (Fig. 2-5). A trailer contains a CTLESC, EOP (End Of Packet) and a checksum (8 bits). The sequence numbers, ranging from 0 to 15, are unique to each outstanding packet. This implies that no more than sixteen packets are out at the same time (section 2.3.2., shows that the number of packets out at one time is only equal to half of the maximum sequence number). The order of these sequence numbers represents the sequence of sending packets (see section 2.3.2).

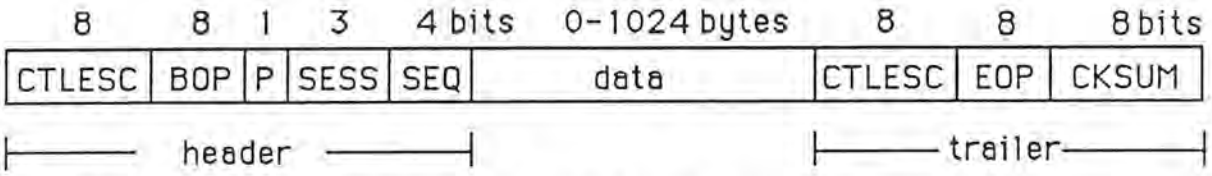

Fig. 2-5 A packet format. CTLESC, BOP and EOP each contain one parity bit.

There are two common ways to determine the size of a packet: a special character at the end of a packet is common among existing protocols, e.g. HDLC and X.25, or having a counter in the header that contains the size of the following data to be transferred, e.g. DDCMP [7, pp.172]. The first method was chosen because of the ease of changing the size of a packet during data transfer. Both methods can provide different packet sizes, but the counter method will limit the size before being sent which may be beneficial for future research on dynamic packets (see section 6).

2.2 State Diagrams

To make the Robin protocol easier to implement, and to have a model to follow, Finite State Machines are used (we refer them by state diagrams afterward). State diagrams are used by LR to separate packets, hot characters and control commands. Input to the input state diagram is data from the serial line. The output state diagram receives its input data from UR. The output of input state diagram and output state diagram is sent to UR and serial line, correspondingly.

2.2.1 Modes

A mode is a conceptually dominant set of states in a state diagram. There are three modes associated with each state diagram: COMMAND, HOT and FILE. In each mode, the corresponding format is expected to be received or sent, i.e. packets are received or sent in FILE mode. In HOT mode, hot characters are sent directly without INTESC in front of each of them. In FILE mode, a hot character can be sent by inserting INTESC before it. In COMMAND mode, only

control commands are received or sent. A control command sequence can be sent in any mode of a state diagram.

2.2.2 Sessions

Robin can support a maximum of eight logical channels over a single physical line. Each session is associated with one logical channel. The session number in a packet header is used to determine which session it belongs to. Session 0 is dedicated to remote log-in, the rest of them are used for sending or receiving one file, but not both at the same time in the same session. The sharing of the line is on a demand basis. If only one file transfer session is active, all packets passing along the line will have the same session number.

2.2.3 Input State Diagram

Mode	State Name	Full name
COMMAND	CMDM	command mode
	C_RESC	received escape character
	C_RCMD	received command
HOT	HOTM	hot mode
	H_RESC	received escape character
	H_RCMD	received command
FILE	FILEM	file mode
	F_RESC	received escape character
	F_RCMD	received command
	F_IFH	interrupt for hot character
	CKSUM	checksum expected
	SEQNUME	sequence number expected

TABLE 1.

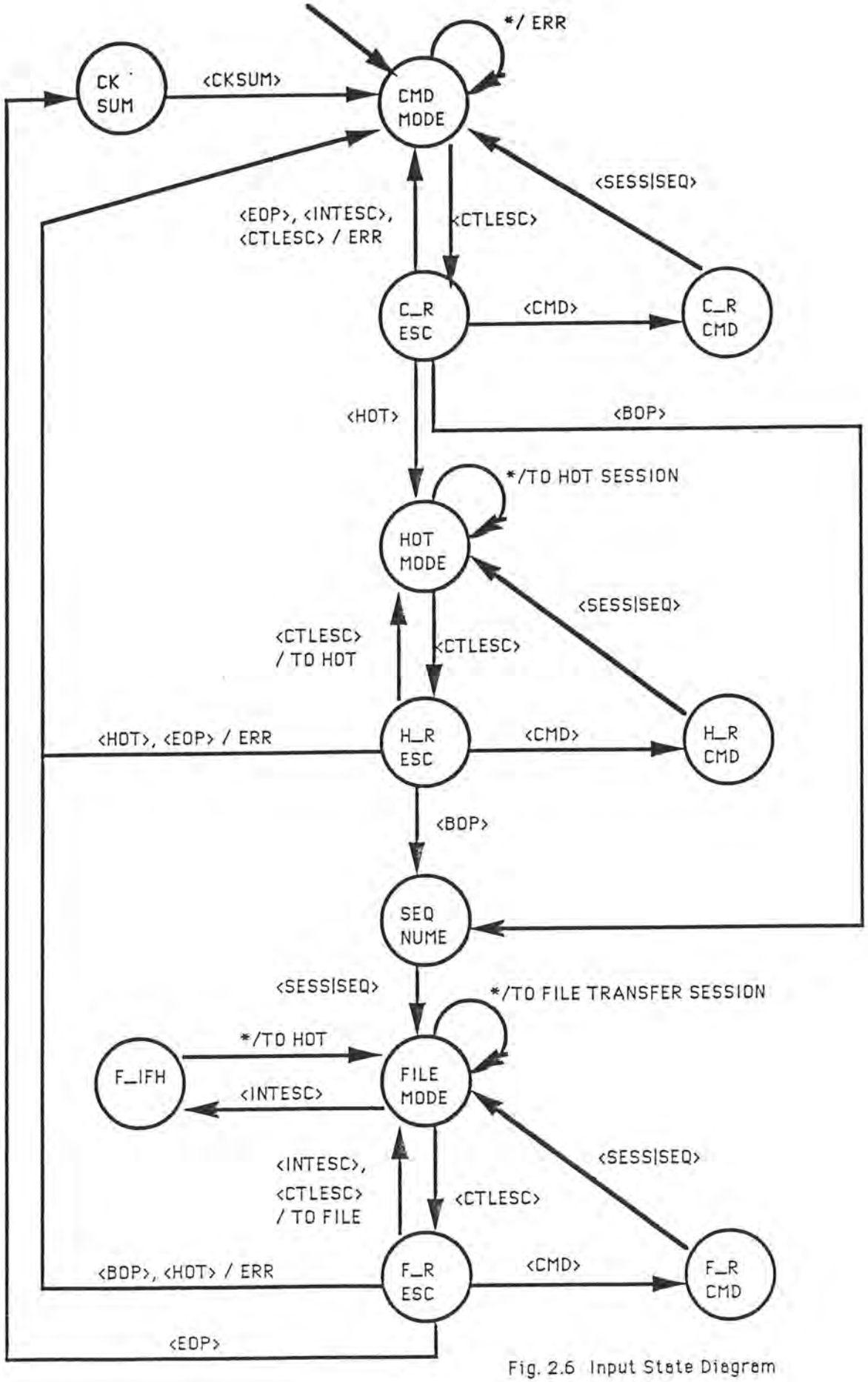

Fig. 2.6 Input State Diagram

The Input State Diagram (Fig. 2-6) consists of twelve states. Table 1 contains the names of all states and their explanations. Notation details are in Table 2.

The initial state is CMDM. While in this state, the protocol is waiting for a control command, a packet, or an instruction (i.e. a sequence of CTLESC HOT) to go into hot mode. After receiving a CTLESC, it goes into C_RESC, and can either accept a control command or a packet. If a command sequence is received, it passes it up to the UR and goes back to CMDM. No data will be transferred in command mode until the protocol explicitly receives control commands which lead to hot mode (CTLESC HOT) or file mode (CTLESC BOP) . If there are any illegal combinations of commands, it sends an error message to the UR and returns to CMDM state.

In HOTM state, every character which comes in is treated as "hot", and is passed up to the UR without any interpretation. If a CTLESC is received, it goes into H_RESC, and can either accept a control command or a packet. If double CTLESC are received, it treats CTLESC as data. If CTLESC BOP (Beginning Of Packet) is received (i.e. a packet is being transmitted), it accepts the next byte as the session number and sequence number, then goes into FILE mode.

Name	Explanations
<BOP>	beginning of packet
<CKSUM>	the byte for checksum
<CMD>	control command
<CTLESC>	control escape
<EOP>	end of packet
<HOT>	explicitly change to hot mode
<INTESC>	interrupt escape
<SESS|SEQ>	session number and sequence number
ERR	error patterns

* : any input bytes except those change the current state to other states.

<> : one byte

TABLE 2.

In FILEM state, every character which comes in will be treated as packet data. If an INTESC is received, the next byte will be sent to hot session. The sequence CTLESC CTLESC and CTLESC INTESC represents CTLESC and INTESC as data. When an EOP (End Of Packet) is received after a CTLESC, it accepts one more byte as the checksum of that packet, then goes back to CMD mode.

2.2.4 Output State Diagram

The Output State Diagram (Fig. 2-7) consists of three states. Each mode contains only one state. Table 3 contain the names of all states and their explanations. Notation details are in Table. 4.

Mode	State Name	Full Name
COMMAND	CMDM	command mode
HOT	HOTM	hot mode
FILE	FILEM	file mode

TABLE 3.

The initial state is CMDM. In this mode, the protocol is waiting for something to send. It expects a control command, hot characters, or packets from its UR. When it receives a control command, it frames a CTLESC in front of the command and a combined session number and sequence number at the end and then transmits this new string. When the character read is a hot character, it goes into HOTM state, and inserts an INTESC before that character. When it is a packet, it goes into FILEM state, and inserts CTLESC BOP SESS|SEQ sequence as the header.

In HOTM state, every character received from UR will be sent immediately. A control command can be sent in this state. If CTLESC is part of the data, double CTLESC will be sent.

In FILEM state, all data in a packet will be sent as they are. If a hot character is received from the UR, an INTESC will be inserted before it. Double CTLESC and CTLESC INTESC

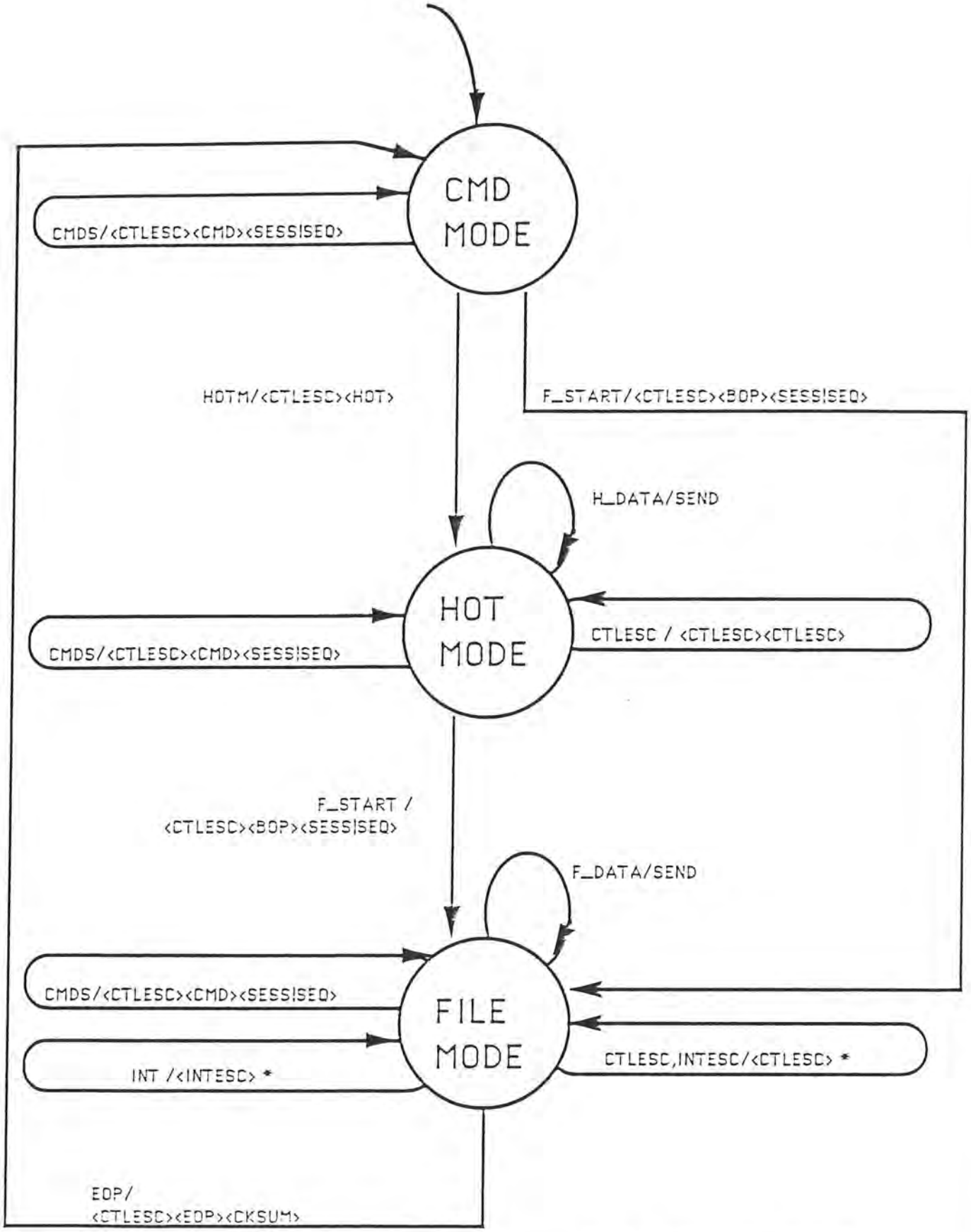

Fig. 2.7 Output State Diagram

represent CTLESC and INTESC as data. A checksum is calculated and sent out after the EOP. After LR completes sending a packet, it goes back to CMDM state.

Name	Explanations
CMDS	control commands from UR
CTLESC	CTLESC as data
EOP	end of packet
F_DATA	file data
F_START	start of file transfer
H_DATA	hot data
HOTM	change to hot mode
INT	interrupt in file mode for hot data
INTESC	INTESC as data

* : a byte that corresponding to the input

<> : one byte

TABLE 4.

2.3 Upper Robin Protocol

The algorithm used in Robin protocol [7, pp.162] is bidirectional, allowing multiple outstanding packets and accepting out of order packets.

2.3.1 Asynchronous vs. Synchronous read/write

Protocols dealing with asynchronous serial lines are usually known as asynchronous protocols because signals passing along this line are asynchronous. This particular form of asynchronization is about the physical line; however, all protocols should be synchronous in transmitting data since the sender and receiver need to be coordinated. Protocols working on half duplex methods have few problems operating synchronously. Kermit [3] is a stop-and-wait protocol [1, pp.143] which sends one packet and then waits for an acknowledgement before

proceeding. Robin can send and receive simultaneously. If there is nothing to read from the serial line, alternate read/write (or synchronous read/write) is not an effective way to handle bidirectional communication. Such a method keeps looking at the line to try to read data even when there is no data to read. Ideally, a transmission line is read whenever there is data, which is known as asynchronous read/write. Synchronous read/write uses a polling technique, asynchronous read/write uses an interrupt technique. Robin uses this interrupt technique to achieve asynchronous read/write (see section 4).

2.3.2 Pipelining and Sliding Window

After a packet is transmitted across a serial line and successfully received, the receiver sends an acknowledgement (ACK) back. The time between sending out a packet and receiving the acknowledgement is known as propagation delay [7, pp.113]. The longer the distance, the longer the propagation delay. Time is wasted while waiting for the acknowledgement as in stop-and-wait protocols, additional packets could be sent. Sending multiple packets before waiting for an acknowledgement is known as pipelining [7, pp.153]. This technique fully utilizes the line.

Let B bits/sec be the baud rate of a serial line, L bits be the packet size, and R seconds be the round-trip propagation time. The time required to send a packet is L/B seconds. It takes R/2 seconds to arrive at the receiver. The receiver takes L/B seconds to read the packet. It takes another R/2 seconds to return an acknowledgement (assuming the time required to read an acknowledgement is negligible). After sending the packet, for stop-and-wait protocol, there is (BR+L)/B seconds idle time. In this idle time, the protocol can send more packets. The number of packets that can be sent to the line before receiving an acknowledgement is (BR+L)/L [7, pp.154]. We assume the packet size (L) and the baud rate (B) are constants. Thus, the longer the propagation time (R), the more packets can be sent to the line before receiving an acknowledgement.

Sliding window [7, pp.151] is a method for keeping track of all unacknowledged packets out on the line. Both sender and receiver have their own sliding windows. Each window is associated with a sequence number and a buffer. If a packet is corrupted during transmitting, the receiver will send a negative acknowledgement (NAK) and continue to accept correct packets, and store them into individual buffers until the sliding windows are full. Whenever these packets are in order again, they will be passed to the upper layer, and the sliding windows will be updated.

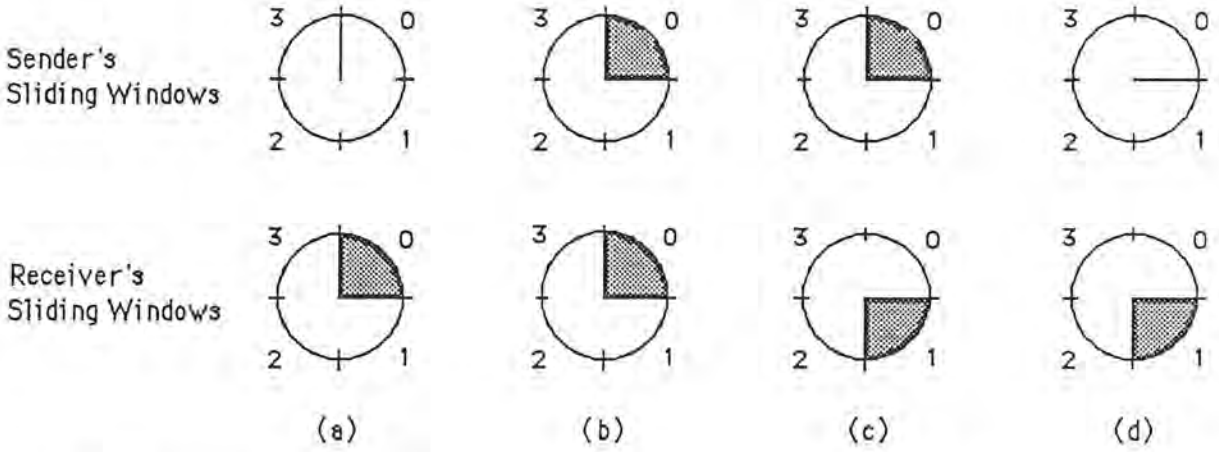

Fig. 2-8 A sliding window of size 4, with 3-bit sequence number. (a) Initially. (b) After the first packet has been sent. (c) After the first packet has been received. (d) After the packet acknowledgement has been received.

It is reasonable to make the size of a sliding window equal to the maximum sequence number + 1. However, Robin protocol can accept out of order packets up to the size of the sliding window, so the size of the sliding window for Robin protocol should be [(maximum sequence number + 1)/2] to avoid overlapping of sequence numbers [7, pp.161]. An example is given in fig. 2-8.

2.3.3 Time Out Mechanism

Using acknowledgement alone is not enough to provide an efficient error detection protocol. If a NAK is lost in the line, the sender must be notified and retransmit the damaged packet. A multiple timers system is adapted and implemented. Each packet is associated with one timer. After sending a packet, the timer is switched on until an ACK of that packet has arrived. If the NAK has been received, the protocol stops the timer and sends that packet again. After retransmitting, it starts the timer again. If there is no acknowledgement at all and the time is out, it retransmits that packet and starts a new timer. The multiple timers system is simulated in software by using a single hardware clock that causes interrupts periodically.

2.3.4 Dynamic Buffering

Since there is no fixed packet size, a sequence of packets may differ in size from each other. This condition leads us to use a dynamic buffering method with the sliding window in fig. 2-9. This technique is more flexible and saves space. The flexibility provides dynamic adjustment of packet size for one particular file transfer session (see section 6).

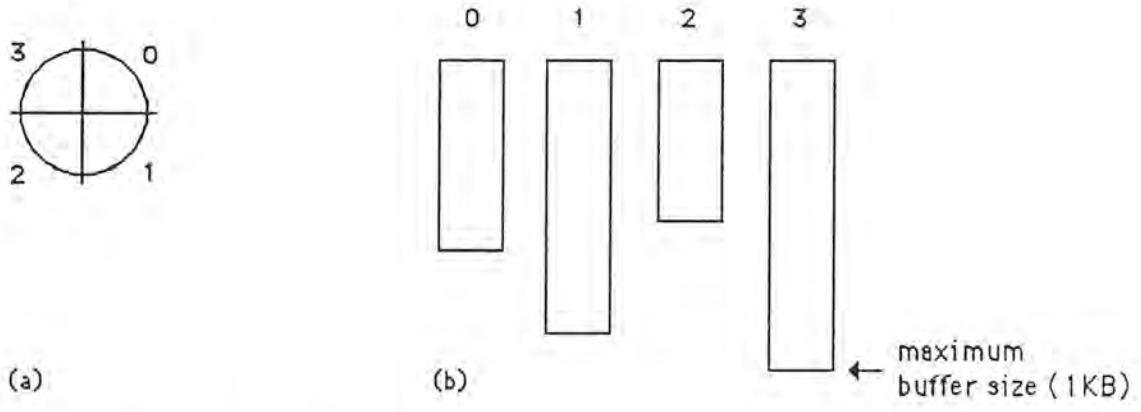

Fig. 2-9 (a) a sliding window with size of 4. (b) Each buffer corresponds to a window with different size.

2.3.5 Checksum

There are a number of methods to calculate a checksum. The most reliable and common is cyclic redundancy code (CRC) [7, pp.129]. However, we chose a simple method: the summation of all bytes sent or received to form an 8-bit checksum. The carry of checksum after overflow is ignored.

3. Robin, SLIP and Kermit

The following brief discussion is based on the designs of Robin, SLIP (see Appendix B or [5]) and Kermit [3]. This is not a complete comparison of these protocols, but a discussion of the main concepts.

(a) Layering

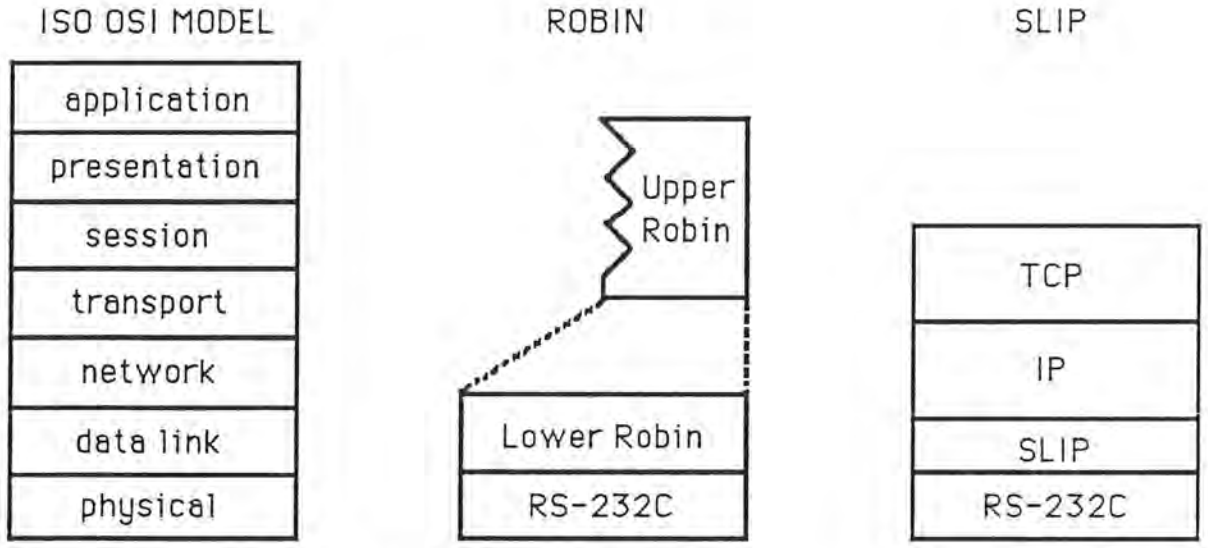

Fig. 3-1 Layers of ISO OSI Model, Robin and SLIP architecture.

Fig. 3-1 is a diagram comparing the ISO OSI Reference model, Robin and SLIP. SLIP is a single layer protocol while Robin and Kermit are similar with multiple layers. By comparing them with the OSI Model, we can get some ideas about functionalities of Robin and SLIP.

Robin is an application and can be used alone. Upper Robin (UR) handles file transfer, multiple sessions and remote log-in. Lower Robin (LR) sends and receives data based on input and output state diagrams. It is quite similar to Kermit, but Kermit doesn't handles multiple sessions at the same time. SLIP is no more than just a simple framing protocol.

(b) Implementation Level

Robin has been implemented in the C language on the user level. It is more portable since it is less dependent on a machine. SLIP is a device driver, so it is closely tied to the operating system. Although it is less portable, it has the efficiency advantage that operating system routines can provide. SLIP has been integrated into 4.3 BSD Unix. Since we wish to develop an interface which will eventually be used with different operating systems and varied hardware, portability seems more important than efficiency. Kermit offers successful trade off.

(c) Error Detection

SLIP is a data link layer without error detection. The layer above SLIP is expected to check by itself any error which occurred in a packet. A protocol which assumes the lower layer to be reliable cannot be used over SLIP. The error detection capability of Robin can provide more general and reliable services to upper level protocols, that means one protocol can provide different services to different upper level protocols.

There are different degrees of error checking. The algorithm chosen for Robin [7, pp.162] accepts out of order packets, but buffers them up until they are in order. Each outstanding packet is

associated with a timer. When a timer of a particular outstanding packet goes off, only that packet is retransmitted, not all the outstanding packets. Because of buffering on the receiver, accepting out of order packets becomes possible. This method is known as a sliding window protocol (see section 2.3 for details).

(d) Data Format

There are hot character, control command and packet formats which can be sent across the line. Section 2.1 has more details on the definitions and usages. However, SLIP only has packet format which is quite passive and has no control over the line. The control commands of Robin provide some control over the line, as well as, more services to the upper level protocols. Kermit is similar to Robin, but has no hot characters.

(e) Addressing

The emphasis of protocols is point-to-point asynchronous serial connection. There is a dedicated serial line or phone line, so addressing is not important and header space can be reduced. If we aim at internetworking capability, addressing is necessary.

(f) Data Compression

Although data compression techniques are good methods to use on slow serial lines, they are not included in these protocols. A data compression algorithm has been suggested for implementation in SLIP. A later version of SLIP may have it. Even if SLIP includes this technique, the big overhead of TCP/IP is still a major problem. However, any method that reduces the big overhead and fully utilizes the line is good for Robin.

4. Discussion on Implementation

Robin has been partially implemented on a Unix operating system. A dedicated serial line was connected to two serial ports on a single machine (VAX 750 running 4.3 BSD Unix). Protocol 6 [7, pp.162] has been modified and implemented to fit our Robin design. File transfer capability and interruptibility have been primarily tested.

There are limitations in the available hardware. This simple experiment used just a short serial line (about 3 feet long). It will hardly notice the propagation delay during transmission. Robin will be implemented with both client and server in the same operating system first, before being ported to a microcomputer. One of the target operating systems is Unix (see Appendix A). Since, it is not quite suitable for what we have designed, Robin has only been partially implemented.

4.1 Implementation Problem under Unix

The problem is the implementation of asynchronous read/write on the user level. In Unix, "signal" is the only way to communicate between the kernel and a user process asynchronously. An I/O signal has been set up so that whenever data is available on the serial line, it receives a signal from the operating system. With this set up, Robin protocol can send data and read data when an I/O signal is received (interrupted). Ideally, the protocol can handle a full duplex line. Theoretically, this technique should work, but practically, the data comes so fast that it fills up the internal buffer of Unix. The rest of the data is discarded. The context switching between the Robin process and kernel process takes time when I/O signals are sent and received. Therefore, there is data lost in the middle of a file transfer. One way to solve this is to read more characters at a time and put them into a buffer by themselves. This approach works fine to clean up the internal buffer overflow problem, but it has a problem of missing characters at the end of a file. It is an inconsistent bug, which, although not guaranteed to occur, has not yet been solved.

4.2 Temporary Solutions

The "signal" approach cannot achieve asynchronous read/write; therefore, in order to test the rest of our design, we used an alternative approach: synchronous read/write. The deficiency of this approach is that we must read, even if there is no data available. This implementation worked and the underlying Robin protocol has been tested using it. There are several alternatives to the synchronous read/write approach. The following sections discuss some of the possibilities.

4.2.1 Xinu Approach

Xinu [1] is an operating system created by Comer at Purdue University. It is mainly used for teaching how an operating system is implemented. Later, Comer implemented a language called Concurrent C which has the same concepts as Xinu, and is a superset of the C language. It provides context switching within the program itself. The Lower Robin has been implemented in Concurrent C, but because of the lack of control over the frequency of the switching, the switch between read and write occurred too often which actually slowed down the whole process.

4.2.2 Non-Blocking I/O

Unix provides non-blocking input/output (see fcntl in [8]) which returns control to the program from the read system call if there is no data available. If non-blocking I/O is set, the read system call will be blocked until there is data. Therefore, the program has to read the line regularly to check whether there is data or not. This approach works, but there is another system call which is more general.

4.2.3 Select System Call

There is a system call called "select" [8] which is used for synchronous I/O multiplexing.

More than one file descriptor can be used. Select will check whether any file descriptor can be read or written. The difference between using "select" and "signal" is that "select" actively checks the availability of data while "signal" is passive. Select has to be used inside a loop so that all I/O will be monitored. The result of using select system call is better than previous methods. However, the problem of missing characters at the end of files still exists but occurs less often.

5. Conclusion

From the Robin design and implementation, we have encountered difficulties in implementing Robin on the user level, especially with the asynchronous I/O part. The problem mentioned in section 4.1 continues to exist. The design of signals in Unix was not intended to be used so intensively for asynchronous communication between a user process and the operating system. The timing problem under Unix is a well-known competitive condition. To circumvent the problem, besides switching to synchronous I/O, we can implement Robin as a device driver (like SLIP did). Then, it will be faster and will allow better manipulation of internal buffers.

6. Future Research

Continuing to develop Robin's potential is worthwhile, although some difficulties have been encountered. The existing SLIP does not solve any problems about big overhead over serial lines. The overhead of Robin is 6 bytes per one character in packet mode for the worst case. There is no overhead in hot mode that will speed up the transmission. However, there is always a trade off between faster transmission and more reliable protocol. With the dynamic adjustment of packet size (section 2.3.4), the priority of a session can be changed by changing the packet size, or vice versa.

Appendix A : Hardware Requirements

Two requirements must be met: the more powerful host should be a Unix-based system, with LAN and modem, and mass secondary storage, and the microcomputer should have a disk drive, a serial port and a modem. The other requirement is a standard RS-232C serial line and a modem set at 9600 baud, no parity, full-duplex, 8 bits data, with one start bit and one stop bit.

Appendix B : Serial Line Internet Protocol (SLIP)

This summary is based on the RFC 1055 (June 1988) [5]. The motivation behind SLIP is quite simple. It encapsulates TCP/IP packets for asynchronous serial line communication. It is currently a de facto standard for point-to-point serial connections running TCP/IP. It has been implemented in 4.3 BSD Unix, Ultrix, Sun Unix and most other Berkeley-derived Unix Systems. SLIP is merely a packet framing protocol. It provides no addressing, error detection/correction, packet type identification or compression mechanisms. A sequence of characters that frame IP packets on a serial line is defined, and nothing more. SLIP is commonly used on dedicated serial links and sometimes for dialup purposes, and is usually used with line speeds between 1200 bps and 19.2 Kbps.

There are 4 special characters which are used to escape the next character or to indicate the end of a packet. They are FRMEND, FRMESC, M_FRMEND and M_FRMESC.

Name	Octal	Description
FRMEND	0300	Frame End Character
FRMESC	0333	Frame Escape Character
M_FRMEND	0334	Meta Frame End Character
M_FRMESC	0335	Meta Frame Escape Character

SLIP Packet			FRMEND
IP header	TCP header	optional TCP data	

Since there is no header for a packet, any data sent is packet data. The FRMEND special character is used to terminate a packet. The data stream looks like the following sequence:

. . . data . . . FRMEND . . . data . . . FRMEND . . . data . . . FRMEND . . .

Data between two FRMEND characters is a packet. Therefore, to send a packet, SLIP simply starts sending the data in the packet. If a data byte is the same code as the FRMEND character, a two byte sequence of FRMESC and M_FRMEND is sent. If it is the same as the FRMESC character, a two byte sequence of FRMESC and M_FRMESC character is sent. When the last byte in the packet has been sent, a FRMEND character is then transmitted. Because there is no length limit for the packet size, there is no theoretical maximum packet size for SLIP. Using the maximum packet size used by the Berkeley UNIX SLIP drivers is suggested, i.e. 1006 bytes including the IP and transport protocol headers (not including the frame character).

There are several features which would make SLIP more efficient, but they are not implemented. Because of the lack of addressing, the IP addresses of both computers must be known in advance. SLIP has no type field. Thus, only one protocol can be run over a SLIP connection on each side; however, not only the IP packet can be sent across the line. Error detection is not absolutely necessary at the SLIP level because any IP application is expected to detect damaged packets. Since transmitting data over a serial line is slow, packet compression would give large improvements in packet throughput.

There is another SLIP (Serial Line Interface Protocol) [2] which was defined in appendix D of RFC 914. Except for a common name, both SLIP methods are different . So far, no one has implemented the RFC 914 SLIP.

References

[1] Comer, Douglas . : *Operating System Design : The XINU approach.* Prentice Hall, 1984.

[2] Conte, T. M., Delp G. S., Farber D. J. : *A Thinwire Protocol for Connecting Personal Computers to the INTERNET.* Document RFC 914 of DDN Network Information Center, 1984.

[3] Frank Da Cruz & Bill Catchings. : *Kermit: A File Transfer Protocol for University Part I & II*, Byte, June & July 1984.

[4] *ISDN: A Means Towards a Global Information Society.* IEEE Communications Magazine 25 pp.30-5, Dec 1987.

[5] Romkey, J. : *A Non-standard for Transmission of IP Datagrams over Serial Lines: SLIP.* Document RFC 1055 of DDN Network Information Center, 1988.

[6] Ross, Ian : *Toward ISDN: Networking for the Nineties.* Electronics & Wireless World 94 pp. 382+, April 1988.

[7] Tanenbaum, Andrew S. : *Computer Networks.* Prentice Hall, 1981.

[8] *Unix Programmer's Manual, Vol I section 2.* BSD Unix 4.3 release.

[9] *Advanced 4.3 BSD IPC Tutorial.* Unix Programmer's Supplement Document, BSD Unix 4.3 release.

[10] Zimmermann, H. : *OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection.* IEEE Trans. Commun., vol. COM -28, pp.425-432, April 1980.