TR-83-003

具有遠方直接使用能力的智慧型前端機

Design and Implementation of a Front End Processor with Multiuser Remote Login Capability

贊助單位：本研究部份由資訊工業策進會贊助，合約 III-RD-009

執行單位：中央研究院資訊科學研究所

計畫主持人：柯　志　昇

研究助理：劉龍駿、陳省隆、林慶良
潘佑安、康錫頂、呂建春

執行期限：71 年 2 月 1 日～72 年 1 月 31 日

日　　期：中華民國　七十二　年　二　月

中研院資訊所圖書室
3 0330 03 000036 3

0036

序 言

本文之內容旨在描述一俱有多用者遠地登入能力的前端機之設計與製作•本研究計畫係前一研究計畫「俱有多用者編輯能力的智慧型前端機」（參見本所研究報告 TR-81-004 ）之延伸•為了便於讀者閱讀本文，文中第二章和第三章之內容係採自前一計畫之研究報告，以求貫通全文•

執行本研究計畫期間，同一研究小組亦進行計算機網路與分散式資料庫系統之研究計畫（參見本所研究報告 TR-83-002 ）兩個研究計畫之部份程式設計有相輔相成之效，有興趣之讀者宜予同時參閱兩份研究報告，更能體會本研究之目的與成果•

本研究計畫承資訊工業策進會資助部份研究經費，特此申謝•

Contents

Design and Implementation of a Front End Processor with Multiuser Remote Login Capability

Jyh-Sheng Ke, Lung-Chun Liu, Hsing-Lung Chen, Ching-Liang Lin
Yo-An Pan, Shyi-Ting Kang and Chien-Chun Lu
Institute of Information Science
Academia Sinica, Taipei, Taiwan, R.O.C.

1. Introduction

This report describes the design and implementation of a Z80-based front end processor with multiuser remote login capability. The purpose of working for this project is to study the economical advantage of using the line concentration technique to reduce the communication line cost of a computer environment and to study the technique of implementing computer communication software programs.

Figure 1.1 shows the computer environment of a conventional computer system. In the conventional system each terminal is directly connected to the host computer, which not only redundantly occupies the computer's I/O interface ports but also requires multiple communication lines. Since that most of time the terminal is in idle status(to wait for I/O data), we may consider to share an interface port and a communication line with many terminals under the control of a low-cost concentrator. Feasibility study shows that it's cost effective to use a microprocessor to function as a line concentrator. Figure 1.2 shows a host computer with a front end processor(FEP) which can function as a line concentrator. In this system, the host computer is a PDP-11/70 with RSX-11/M operating system, and the FEP is a Z80-based microcomputer.

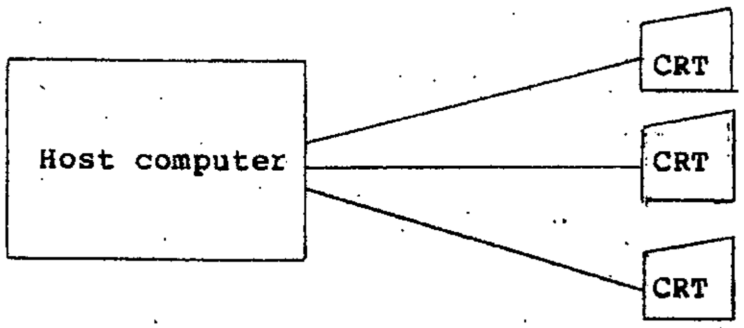


Figure 1.1 Conventional computer system

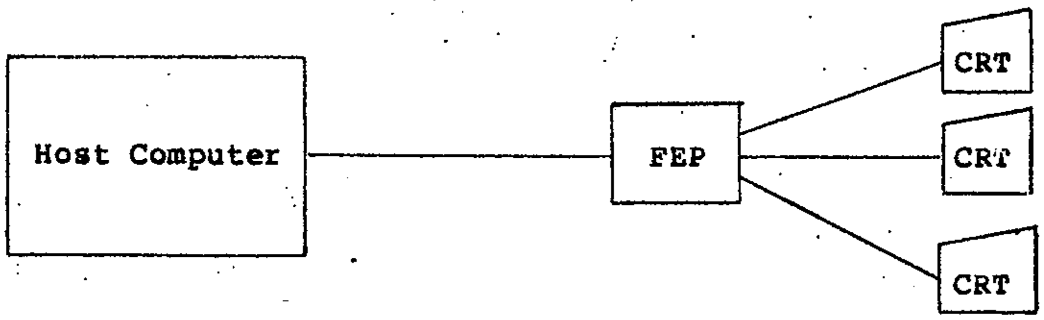


Figure 1.2 A computer with front end processor

To accomplish the functional requirements of the FEP, at least the following things have to be done:

- implementation of communication software programs and terminal handlers in FEP.
- implementation of software programs for multiplexing and demultiplexing data in RSX-11/M system.

. implementation of peer-protocol machine programs for handshaking between FEP and the host computer.

To fulfill these requirements, we have implemented a real-time event-driven multitasking executive in FEP for coordinating a set of peer-protocol machine programs and the terminal drivers. We also have implemented a set of peer-protocol machine programs in RSX-11/M system by employing the concept of virtual terminal to achieve the multiplexing and demultiplexing functions. The communication software programs can also be used to talk with a local area computer network[14].

The main implementation problem when connecting RSX-11/M to an FEP or a network is where to site the protocol handler. It can either be installed as part of the terminal I/O driver; or it can be run as a system task communicating with the network through some interface to the terminal I/O driver. Due to the complexity of the original RSX-11/M terminal I/O driver, it was decided that the protocol handler must be run as a background task and communicating with the network through an add-in virtual terminal driver as an interface to the terminal I/O driver. Figure 1.3 shows the terminal I/O driver function of the standard RSX-11/M ststem. Figure 1.4 shows the modified RSX-11/M in which three additional software modules have been incorporated to function as a virtual terminal protocol handler via which multiple terminals can be connected to the PDP-11/70 through a Z80-based concentrator. In the following, Section 2 describes the hardware configuration of the system, Section 3 presents the implementation of a real-time event-driven multitasking executive, Section

4 describes the software program modules and protocol machines in the FEP, and Section 5 describe the design and implementation idiosyncracies of the communication software programs in the RSX-11/M system.

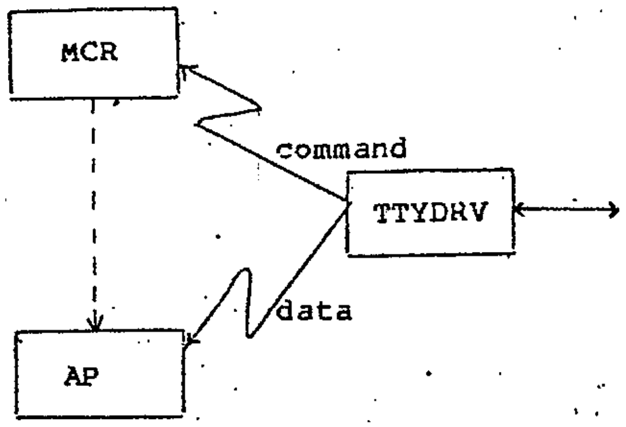


Figure 1.3 Standard RSX-11/M Terminal I/O

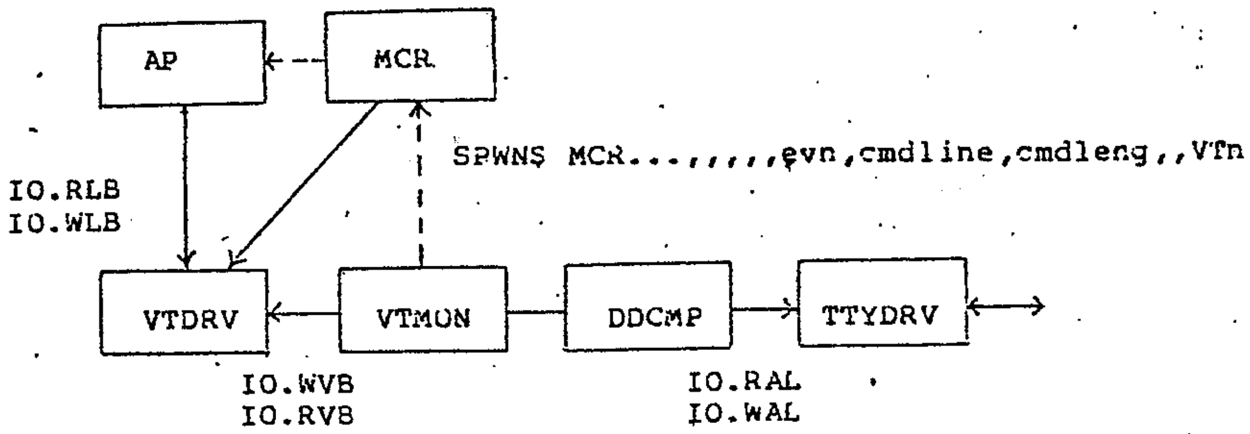


Figure 1.4 Modified RSX-11/M Terminal I/O

2. Hardware Configuration

As a front end processor, a system with a minimum hardware cost appears to be fitting a basic requirement. In the FEP with local editing and remote login capability, the minimum hardware configuration with only two functional boards are used. In this section we will describe their modular functions.

MCB (the Zilog Z80-MCB) is a single board microcomputer designed to be adaptable to a wide range of OEM applications. The block diagram on the Figure.2.1 identifies the major components on the board. The heart of which is the Z80 microprocessor. Associated logic includes 4K bytes of dynamic RAM, provision for up to 4K bytes EPROM, both parallel and serial I/O ports, I/O ports decoders and a crystal controlled clock. The parallel I/O port is implemented with the Z80-PIO with an area reserved for user applied driver and/or receiver logic. The Z80-CTC is used as a baud rate generator for the serial interface implemented with an 8251 USART.

SIB(the Z80-SIB) uses four 8251 USART devices to implement the serial communication channels. Two Z80-CTC devices are used to accommodate Z80 interrupt capability for receivin and transmitting operation of each bi-directional serial channel. The third Z80-CTC device is provided to accommodate programmable baud rates for each serial port from 50 to 9600 baud, derived from an on-board crystal, the system clock, or an external clock.

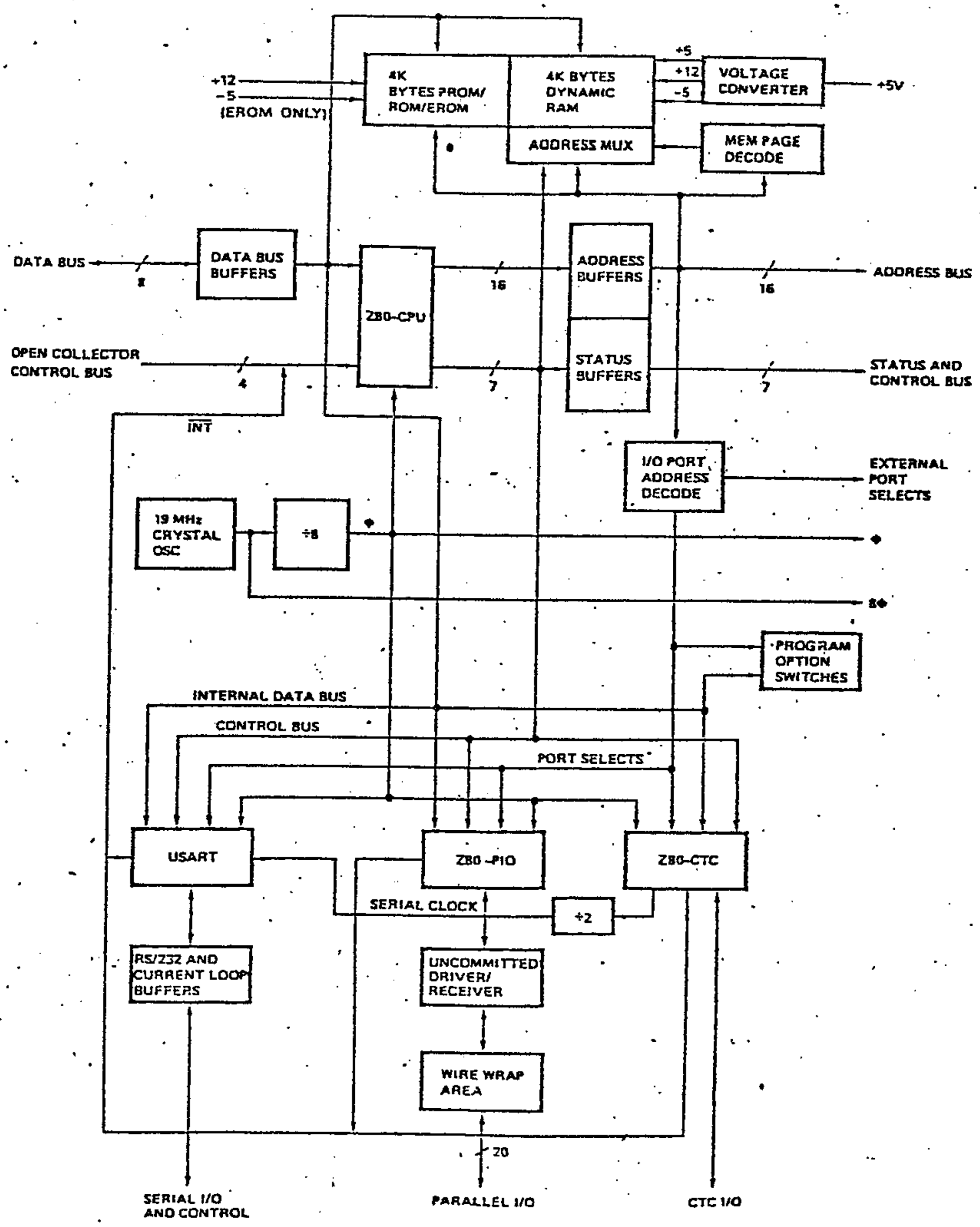


Figure 2.1 Z80-MCB BLOCK DIAGRAM

2.1 Hardware Modifications

There are several modifications over the board from its original deliveries to fit our own system configuration. The primary modifications are through the 'jumpers' to select memory mapping and I/O addressing.

(1)MCB:

.RAM memory jumpers

We have changed the 4K RAM capacity to 16K RAM capacity

J1-10/J1-12,J1-9/J1-16,J1-8/J1-11

.RAM page decoding

The RAM is placed on the hex addresses 4000 - 7FFF.

J3-3/J3-9,J3-11/J3-12

.ROM page decoding

The ROM is placed on the hex addresses 0000 - 3FFF.

J3-3/J3-15,J3-13/J3-16

.serial interface interrupt jumper

The serial interface in MCB is used to communicate with the host, it will generate interrupts when any receive or transmit operation is occured.

J1-13/J1-1

.serial interface jumpers

The serial interface mode is selected as follows:

RS-232-C, MCB='MODEM', always 'clear to send', ignore 'request to send', signal swings from +12V to negative supply.

J4-1/J4-2,J4-9/J4-10,J4-5/J4-6,J4-14/J4-15,

J4-12/J4-11,J4-4/J4-7/J4-16,J5-1/J5-8

(2)SIB:

.port address range selection

The port address range is selected in the hex 80 to 9F

J4-5/J4-16,J4-1/J4-7,J4-4/J4-6

.port of device selection

device	J1 jumper	port address
CTC0	1 - 12	80H - 83H
CTC1	2 - 11	84H - 87H
CTC2	3 - 10	88H - 8BH
USART 0	4 - 14	8CH - 8DH
USART 1		8EH - 8FH
USART 2	5 - 13	90H - 91H
USART 3		92H - 93H

.baud rate generation

The on board PHI/2 is used as clock source

J3-6/J3-11/J3-12/J3-13

.USART clock input

CTC CLK0 drives the TxC0,RxC0,TxC1,RxC1

J2-2/J2-5/J2-6/J2-13/J2-14

CTC CLK1 drives the TxC2,RxC2 J2-3/J2-15/J2-16

CTC CLK2 drives the TxC3,RxC3 J2-4/J2-1/J2-12

3. A Real-Time Event-driven Multitasking Executive

In order to accomplish the multiuser facility, a multitasking architecture is required to facilitate the software design. Different modules can be separated into independent tasks, with a minimum coupling during system design stage. Moreover, we can assign different tasks to different users, in which different tasks may share the same program. In this system, we have designed a simple but useful executive, called real time event-driven multitasking executive(REMX), to provide a multitasking environment.

3.1 General concept of the REMX

A task in the system is an independently executable program. Associated with each task is a task control block (TCB), which is used to maintain control information about the task, such as the program entry point, stack pointer, event control word, and some pointer fields about messages. Each task can be in one of the three states, namely RUN, READY, and BLOCK (WAIT). The state of a task is stored at the task status word (TSW) in TCB.

The communication between tasks is through the "message exchange channel". As in figure 3.1, one task may send a message to a channel , on the other end the task that requires this message can receive it from the same channel. The channel structure provides a many-to-many relationships among tasks. This structure seems to be more useful, for example, there may be two spooling

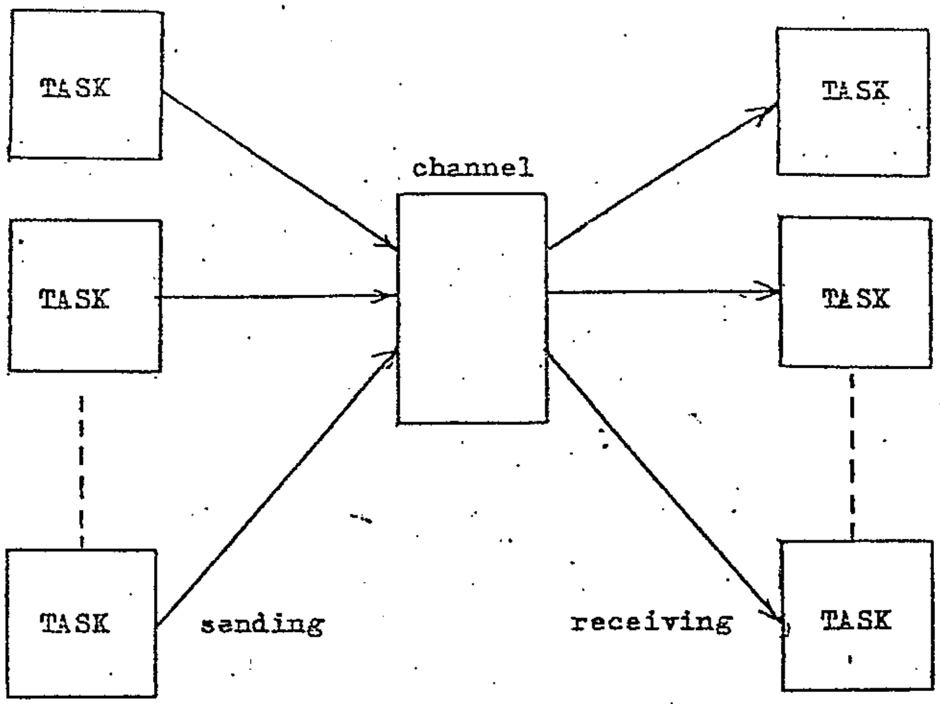


Fig. 3.1 Message exchange channel

output control tasks, any task wants to have its output be spooled need only send a request message to a specific channel regardless which spooler task will process it. There are two types of channels in our system, one is the software channel for inter-task communication and the other is the I/O channel for task-device communication. A task may send a message to a channel or receive a message from a channel. This is illustrated in figure 3.2 (a) and (b).

The synchronization between tasks and I/O devices is through an "event flag" mechanism. A task may send a message to a channel and declare an event flag for synchronization. By associating task operations with event flags, several operations can proceed concurrently and may be synchronized by the mechanism of event flags. Upon the occurrence of an event the processing of the

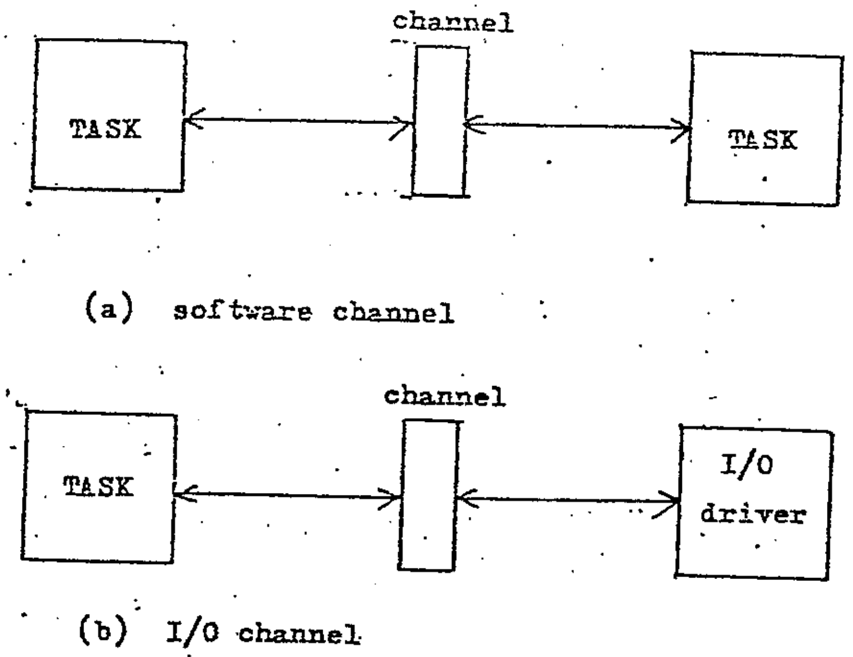


Fig. 3.2 Two types of channel

current task may be continued or discontinued , depending on the dynamic requirements of the system.

There are five functional modules contained in the REMX :

(1) the task scheduler,

(2) the memory manager,

(3) the inter-task communication manager,

(4) the input/output communication manager,

(5) the real time manager.

The task scheduler manages the overall system, schedules tasks, and keeps track of the task status. The memory manager handles the memory allocation and deallocation. The two communication modules control the flow of messages among tasks. The real time manager maintains the real time clock and allows the creation of events based on timers. Fig.3.3 shows the interrelation

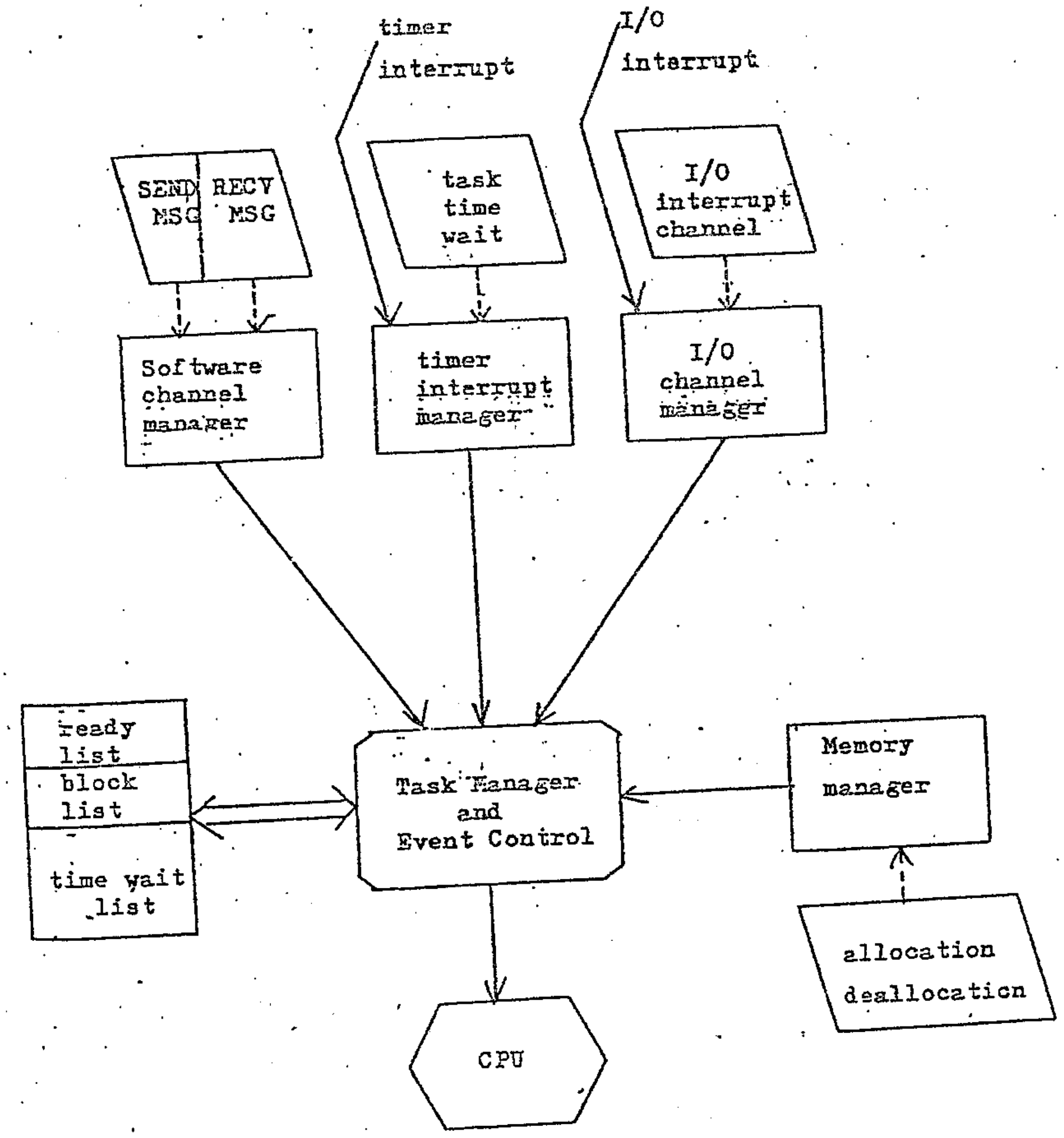


Fig. 3 .3 Interrelation among modules

among these modules.

There are several system primitives that users can issue them to activate system functions. Fig.3.4 illustrates these system primitives.

.TWAIT:	wait a specified time slice
.MARKT:	mark a time event flag
.WAITE:	wait an event flag
.CMRT :	cancel previously marked timer
.SEND :	send a message from a channel
.SENDW:	send and wait until the other part received
.RECV :	receive a message from a channel
.RECVW:	wait until a message arrived
.SIGNL:	signal an event flag
.IO ;	issue an I/O request
.IOW ;	issue an I/O request and wait until I/O complete
.CNRIO:	cancel the previously issued I/O request

Fig. 3 .4 System primitives

3.2 The data structures of REMX

The whole system, under the monitoring of the REMX, has three basic data structures to achieve multitasking, channel communication, and message exchange facilities. Each of them, task, channel and message, is associated with a control block to maintain its relevant informations, namely the TCB, the CCB, and the MCB. In this section the data structures will be presented.

(1) TCB (Task Control Block):

```
word offset      TCB
           |--------------------|
   Ø       |      TCBLINK       |
           |--------------------|
   1       |        ISP         |
           |--------------------|
   2       |        IPC         |
           |--------------------|
   3       | TSKNO   |   PRIO   |
           |--------------------|
   4       |     reserved       |
           |--------------------|
   5       |      STATUS        |
           |--------------------|
   6       |     CCBLINK        |
           |--------------------|
   7       |     MCBLINK        |
           |--------------------|
   8       |       ECW          |
           |--------------------|
   9       |       EBW          |
           |--------------------|
  1Ø       |     reserved       |
           |--------------------|
```

TCBLINK:	link pointer to other TCB with same priority
ISP :	initial stack pointer
IPC :	initial program entry point
TSKNO :	task number
PRIO :	priority level number
STATUS :	task status word
CCBLINK:	link to CCB while block for channel
MCBLINK:	link to MCB while receive a message
ECW :	event control word, 1 bit per event
EBW :	event block word, 1 bit per event

(2)CCB (Channel Control Block):

```
    word offset     CCB
                  |-----------------|
        0         | NOMSG  | NOTSK  |
                  |-----------------|
        1         |    LINKHEAD     |
                  |-----------------|
        2         |    LINKTAIL     |
                  |-----------------|
        3         |    reserved     |
                  |-----------------|
        4         |    INTPREAM     |
                  |-----------------|
        5         |    INTCOMPL     |
                  |-----------------|
        6         |    INTHANDL     |
                  |-----------------|
        7         |    reserved     |
                  |-----------------|
```

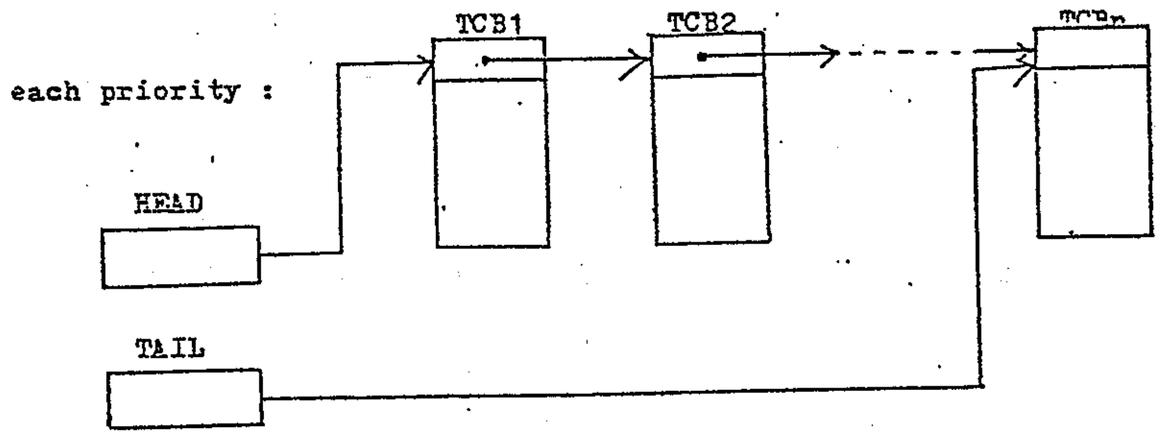
NOMSG	:	number of message available on the channel
NOTSK	:	number of task is waiting for message
LINKHEAD	:	message queue head pointer
LINKTAIL	:	message queue tail pointer
INTPREAM	:	interrupt preamble routine address
INTCOMPL	:	interrupt completion routine address
INTHANDL	:	interrupt handler routine address

(3)MCB (Message Control Block):

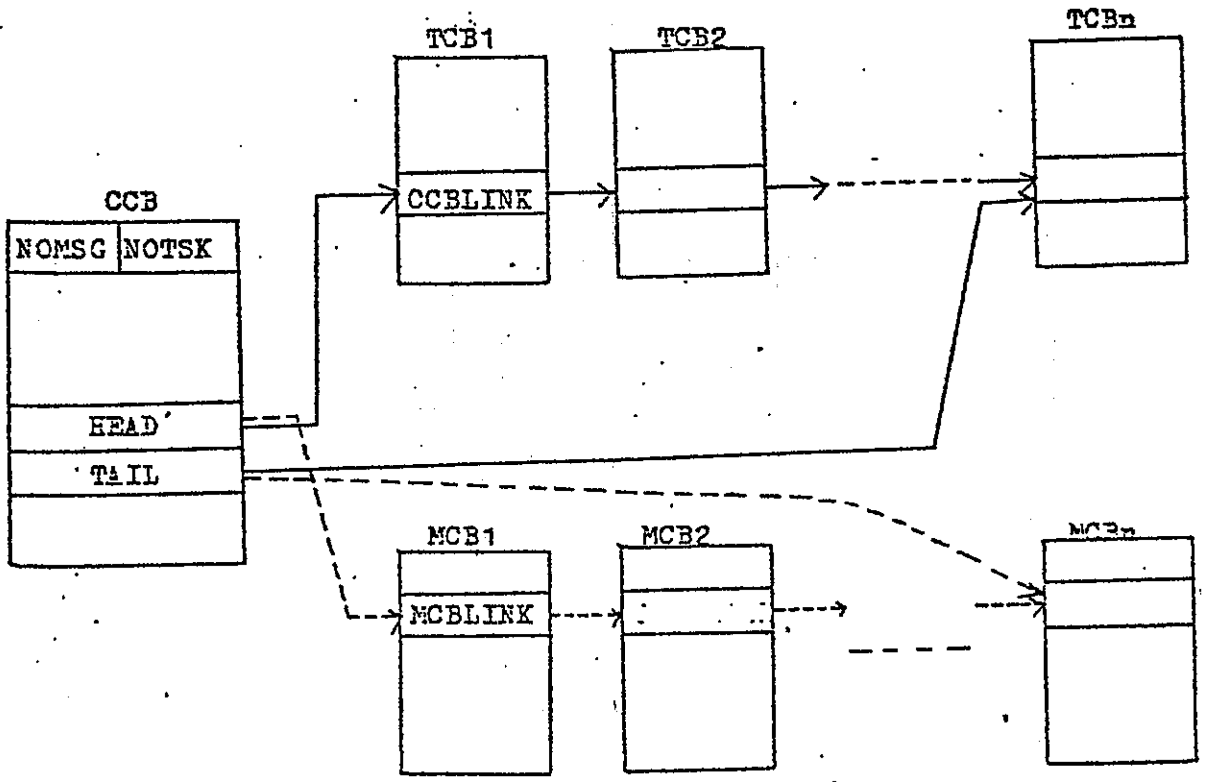
```
word offset      MCB
            |*****************|
     0      |  TYPE  |   ECB  |
            |*****************|
     1      |     MCBLINK     |
            |*****************|
     2      |     TCBLINK     |
            |*****************|
     3      |     MSGPTR      |
            |*****************|
     4      |     MSGLTH      |
            |*****************|
     5      |    reserved     |
            |*****************|
```

TYPE	:	user supplied message type
ECB	:	event control byte
MCBLINK	:	link pointer to next MCB in CCB message queue
TCBLINK	:	link pointer to the sending TCB
MSGPTR	:	message buffer pointer
MSGLTH	:	message length

The relations among these control blocks and data structures are shown in Figure 3.5. For active tasks a ready list is maintained at each priority level. In CCB,a list is also kept to have a TCB queue or MCB queue. If more tasks wait for messages(receivers more than senders), a TCB queue is required. In the case of senders, more than receivers a MCB queue is kept. Because these two cases are mutually exclusive and never occur at the same time, only a list is required. There are two entries contained in each list, one for header pointer and the other for tail pointer. The header pointer is used to remove an item from the list and the tail pointer is used to insert an item into the list.



(a) TCB ready list



(b) CCB, TCB, and MCB

Fig. 3.5 Relations among control blocks

3.3 The Control Flow and Algorithm of the REMX

The software of REMX is structured as the following diagram Fig.3.6.

(a) INIT	:	system initialization routine
(b) TSKMGR	:	task manager
(c) COMMGR	:	inter-task communication manager
(d) TIMMGR	:	timer manager
(e) IOMGR	:	input/output device handling manager

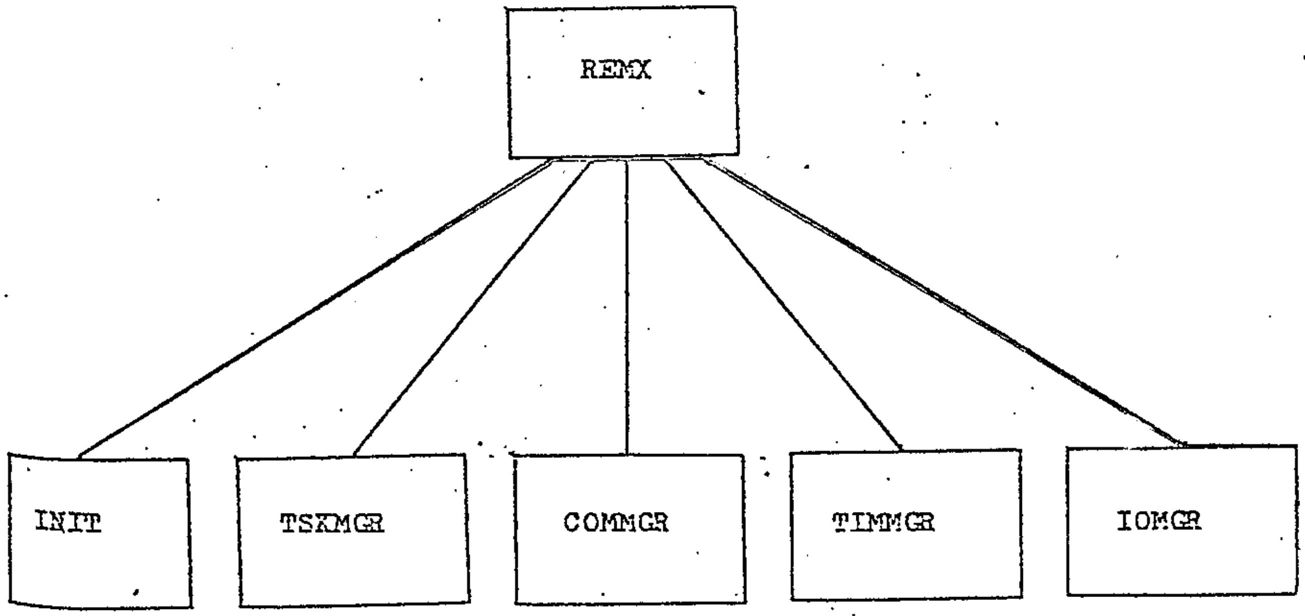


Fig. 3.6 REMX software structure

3.3.1 INIT

The INIT module initilizes the whole system, including system parameters, TCB, CCB and IOCCB.

(1)system parameters and interrupt mode initialization

```
.load system stack pointer
.clear all system parameters to zero
.set all list pointer to -1(null)
.set interrupt mode = 2 (indirect mode)
.set interrupt vector register
.set-up interrupt vector table
```

(2)TCB initialization

```
.get TCB address in TCBD (initial task descriptor)
.if TCB address -1 (null)
    then call BLDTCB to build up this TCB
         call SCHTSK to insert it into ready task
                        queue list
```

(3)CCB initialization

```
.get CCB address in CCBD (initial channel descriptor)
.if channel no. -1 (null)
    then insert CCB pointer into CCBLST
         clear all parameters in CCB
         set all pointers in CCB to -1 (null)
```

(4)IOCCB initialization

```
.get CCB address from IOCCBD (initial I/O channel
 descriptor)
.if channel -1 (null)
     then insert CCB pointer into IOLST
.clear all parameters in CCB
.set all pointers to -1 (null)
.get three I/O process routine address and save in IOCCB
    IO_PRM (preamble routine)
    IO_COM (completion routine)
    IO_HDL (handler routine)
.set IO_HDL in interrupt vector table
```

(5)user's initialization

```
.call user's start routine
```

(6)system start

```
.transfer to dispatcher
```

3.3.2 TSKMGR

The task manager performs the task scheduling, task switching, and the system states entering or exiting.

(1)TSKMG: preempt the current task and reschedule again

```
.save all registers in user's stack area
.change into system state
.if active task is a null task (all tasks are blocked)
    then transfer to dispatcher
.save stack pointer in active TCB
.reload system stack pointer
.check task status word
.if TSW is block
    then transfer to dispatcher
.if TSW is ready
    then call ENQUE to insert the task into ready queue
    if priority > active task priority
       then ACTPRI = priority
.transfer to dispatcher
```

(2)DISPCH (task dispatcher):

```
.for I = MAXPRI (max priority) to Ø (lowest priority)
  do
     if the ready task queue is not empty (not -1 null)
      then
        ACTPRI = I (active task priority)
        SYSPRT = Ø (preemption flag = Ø)
        call DEQUE to deque this TCB from ready queue
        ACTTNO = task number (active task number)
        ACTTCB = pointer of TCB (active TCB pointer)
        reload user's stack pointer
        restore all user's registers
        transfer control to user (dispatch the CPU to use
      endif.
.set ACTTNO = -1 (null task)
     ACTPRI = Ø  (lowest priority)
.reload system stack pointer
```

(3)ENQUE (enque a node into a list)

```
entry condition : HL : queue tail pointer
                  DE : enqued node pointer
.set enqued node's next pointer to null (*1)
.(HL)<*DE save enqued node pointer in queue tail
.save enqued node pointer in the last tail node's
 next pointer
.if the queue is empty before insertion
    then set the queue head = the enqued node
```

(4)DEQUE: deque a node from a list

```
entry condition: HL : queue head
exit condition : DE : the dequed node
.DE<*(HL), get the head pointer to DE
.(HL)<*next node pointer of queue head
.if the queue is empty after deque
    then set the queue tail to *1 (null)
```

(5)ENTSYS: entering system state

```
functions : all system routines must call this routine to set
            into system state and a return address on stack,
            after return from system routine will transfer to
            RETSYS
  .set SYSTAT = 1 (system state flag)
  .push RETSYS onto stack
```

(6)RETSYS: returning from system state

```
functions : after system routine's process there may be some
            preemption condition occurs, so must be checked
            here
  .set SYSTAT = Ø (clear system state flag)
  .if SYSPRM = 1 (any preemption occurs)
      then transfer to TSKMG (task manager)
      else return to user's routine
```

(7)INTSYS: entering interrupted system state

```
.increase interrupt level count
.push DETSYS onto stack
```

(8)DETSYS: returning from interrupted system state

```
.decrease interrupt level count
.if interrupt level count Ø
    then return
    else check system state and preemption flag
        if in user's state and preemption occurs
            then transfer to TSKMG
```

3.3.3 COMMGR

The COMMGR module manages the inter-task communication, task synchronization, message sending/receiving, and event flag mechanism.

(1)SEND: send a message to a channel

```
.call ENTSYS to enter system state
.save event number in MCB_ECB
.get CCB address about the channel number
.increasing message number (NOMSG=NOMSG+1)
.decreasing task number (NOTSK=NOTSK-1)
.if there is a task waiting for message (NOTSK > Ø)
    then if not a send wait request
            then call SETECW to set event control word
         call DEQUE to deque the MCB from the list
         call SCHTSK to insert it in the ready queue
    else call ENQUE to enque the MCB into message list
         save TCB in MCB's sending TCB
         if not a send wait request
            then call SETECW to set event control word
            else set block status
                 transfer to TSKMG
.return
```

(2)RECV: receive a message from a channel

```
.call ENTSYS to enter system state
.call GETCCB to get the CCB address
.NOMSG=NOMSG-1
.NOTSK=NOTSK+1
.if a message is available (NOMSG > Ø)
    then call DEQUE to get a MCB from message queue
      if the message is sent by SENDW
       then call SCHTSK to schedule the sending task
       else return
    else if this is a receive wait request (RECVW)
         then call ENQUE to insert the TCB into CCB
              call BLKCHL to wait until message is available
              return the MCB pointer to user
         else NOMSG=NOMSG+1
              NOTSK=NOTSK-1
              return
```

(3)EVENTF: check event flag condition while a event is occurs

```
.call GETECW to get the event mask
.ifany block condition is met for this event
    then clear EBW in TCB
    else set zero flag
.return
```

(4)SIGNAL: signal a significant event to a task

```
.get TCB number of that task
.calculate index to TCBLST
.get TCB address
.call EVENTF to check event state
.if that task is blocked for this event
    then call SCHTSK to reschedule that task
    else return
```

(5) WAITE: wait for some events and proceed to execute
after any event occurs.

```
.get ECW from TCB
.check event mask with ECW
.if no event has been occured
  then call BLKCHL to wait until any event occurs
  else calculate the event number that has been occurred
       return event number in A to user
```

3.3.4 TIMMGR

The TIMMGR module maintains a system clock, synchronizes the task with time based event, and handles the timer interrupt.

```
(1)TIMGER: declare a event flag under a time basis or wait
           until a time slice has elapsed.

     .calculate index to TIMLST according to the task no.
     .get previous time slice
     .if previous slice = Ø (no timer request before)
         then TIMCNT=TIMCNT+1 (TIMLST count)
     .if a time wait request (TWAIT)
         then call BLKCHL to wait a specified time delay
         else call SETECW to set associated event flag


(2)CMRKT: cancel a previously requested timer event

     .calculate index to TIMLST according to the
      task number
     .get previous time slice
     .if previous time slice Ø (mark time before)
         then TIMCNT=TIMCNT-1
     .if the associated event flag number is not zero
         then call GETECW to get event mask
              clear the event flag
         else return


(3)TMINT: system clock interrupt handle routine

     .save all registers
     .update the system time for
         second, minute, hour, date, and month
     .if a second has been elapsed
         then do I=1 to TIMCNT (scan TIMLST)
                 get time slice
                 if time slice Ø
                    then decrease time slice
                      if time slice count down to zero
                         then TIMCNT=TIMCNT-1
                          call EVENTF to check event flag
                          call SCHTSK to reschedule the task
             end do.
     .restore all registers
     .return from interrupt
```

3.3.5 IOMGR

The IOMGR module accepts the I/O requests from user's, en-ables the I/O channel, handles the I/O interrupt.

(1)IORQS: request an I/O message through an I/O channel

```
.call GETCCB to get IOCCB address
.save ECB in IOMCB
.RQSTNO=RQSTNO+1
.if the I/O channel is not active yet
    then calculate interrupt vector offset in INTVTB
         call user's preamble routine
.call ENQUE to link the new message to message queue
.save TCB address in IOMCB
.clear MSGCNT in IOMCB to zero
.if the ECB < Ø (IOW)
    then call BLKCHL to wait until I/O completion
    else call SETECW to set the associated event flag
.return
```

(2)CNRIO: cancel I/O request on an I/O channel

```
.call GETCCB to get IOCCB address
.if RQSTNO = Ø (no I/O request has been issued)
    then return
    else RQSTNO=RQSTNO-1
         call DEQUE to remove the current IOMCB
         mark an I/O aborted flag on ECB
         if no any more I/O request
            then call user's I/O completion routine
.return
```

(3)INTSV: interrupt save routine

```
.save all registers
.get IOCCB address
.if RQSTNO Ø (some I/O request has been issued)
    then get IOMCB address from IOCCB
      return MCB address to user interrupt handle routine
    else return from interrupt
```

(4)INTEX: interrupt exit routine

```
    .if I/O completion flag is not set
        then return from interrupt
    .RQSTNO=RQSTNO-1
    .if RQSTNO = Ø
      . then call user's I/O completion routine
    .call DEQUE to deque the current IOMCB
    .check the ECB in the IOMCB
    .if ECB < Ø (block request)
        then call SCHTSK to reschedule this task
        else call EVENTF to check event flag condition
    .set preemption flag
    .transfer to TSKMG
```

4. FEP's Functional Modules

Figure 4.1 shows the relationships between REMX and the application tasks under its coordination.

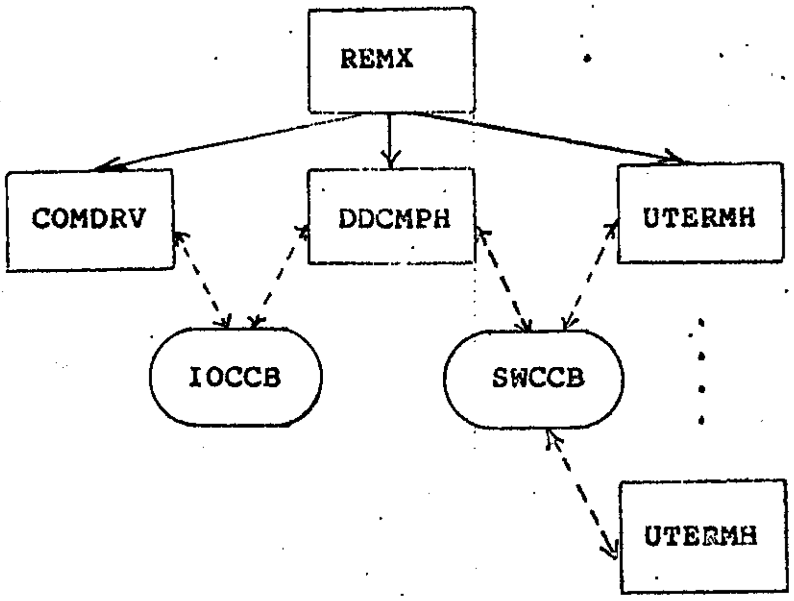


Figure 4.1 Functional modules in FEP

COMDRV is the communication port I/O driver, through which data can be sent to or received from the host computer. DDCMPH is the data link level protocol handler which guarantees the correct frame transmission between FEP and the host by communicating with COMDRV via an I/O channel control block. UTERMH is responsible for the interaction between the terminal user and the FEP. It provides a high-level virtual communication channel between the user and the host computer by communicating with DDCMPH via a software channel control block. More specifically, UTERMH and its peer protocol handler in the host computer cooperatively construct a virtual channel and the associated virtual terminal. Each user terminal is associated with a UTERMH task. In this implementation, the FEP can be connected with four user terminals, and accordingly four UTERMH tasks are created under REMX. However, these four UTERMH tasks use the same program module with each individual's own TCD. This feature reveals the reentrant programming power of the multitasking executive.

In brief, the software program of the FEP mainly implements the layered structure of the protocol handlers(see Figure 4.2).

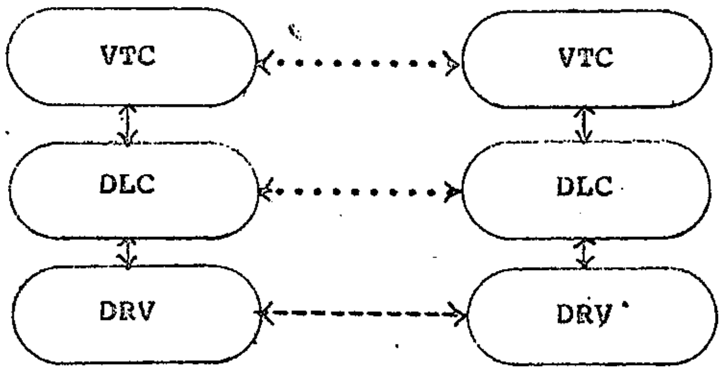


Figure 4.2 Layered Protocol Structure

In the following we will describe the design idiosyncracies of each software module.

4.1 COMDRV -- Communication Port I/O Driver

COMDRV essentially contains an input interrupt service routine(IISR) for handling data input from the host and an output interrupt service routine(OISR) for handling data output to the host.

IISR receives and identifies data frames from the host by the following procedure:

- input a data byte from the port, if this data byte is the header of a frame, then
- collect a frame in the buffer
- if receive error(CRC error) then ignore it, else
- pass the received frame to DDCMPH task by putting the control-frame and/or data-frame in their respective IOCCB.

OISR gets a frame from the associated IOCCB, which is sent by the DDCMPH, and outputs byte-by-byte to the host.

The data frame format and types are illustrated in Figure 4.3.

Information frame

SOH	data count	R#	S#	header CRC	data	frame CRC

data count: length of data in bytes
R# : received data's sequence number
S# : sending data's sequence number
CRC: cyclic redundancy check

Supervison frame

ENQ	TYPE	SUBT	R#	S#	header CRC

TYPE can be one of the following codes:

STRT (22):	request to start
STACK(24):	acknowledge to start
DISC (26):	request to disconnect
ACK (Ø6):	acknowledge data received
NAK (25):	negative ACK
REP (2Ø):	request reply acknowledgement
RST (27):	reset protocol machine

SUBT : reserved for subtype control

Figure 4.3 Message frame formats

4.2 DDCMPH -- Data Link Protocol Handler

The main function of DDCMPH is to provide a perfect virtual channel for two communicating entities through multiplexing techniques. It is a DEC's DDCMP-compatible data link protocol control program. It formats the data from the UTERMH tasks into a frame and sends the data frame to the host via COMDRV. On the other side, DDCMPH receives a frame data(via COMDRV) from the host, examines the frame header, and sends the data frame to the right destination UTERMH task(the reader should be noted that we have four UTERMH tasks).

Presumably the DDCMPH task has to achieve the following functional requirements:

- . multiplexing
- . frame sequencing
- . frame flow control
- . channl connection and disconnection

These functional requirements are achieved by following communication protocols which are illustrated in Figure 4.4 and Figure 4.5. From Figure 4.4 we know that connection between FEP and the host computer is always initiated by the FEP. After successful connection, exchanges of data between FEP and the host computer follows the alternating-bit protocol.

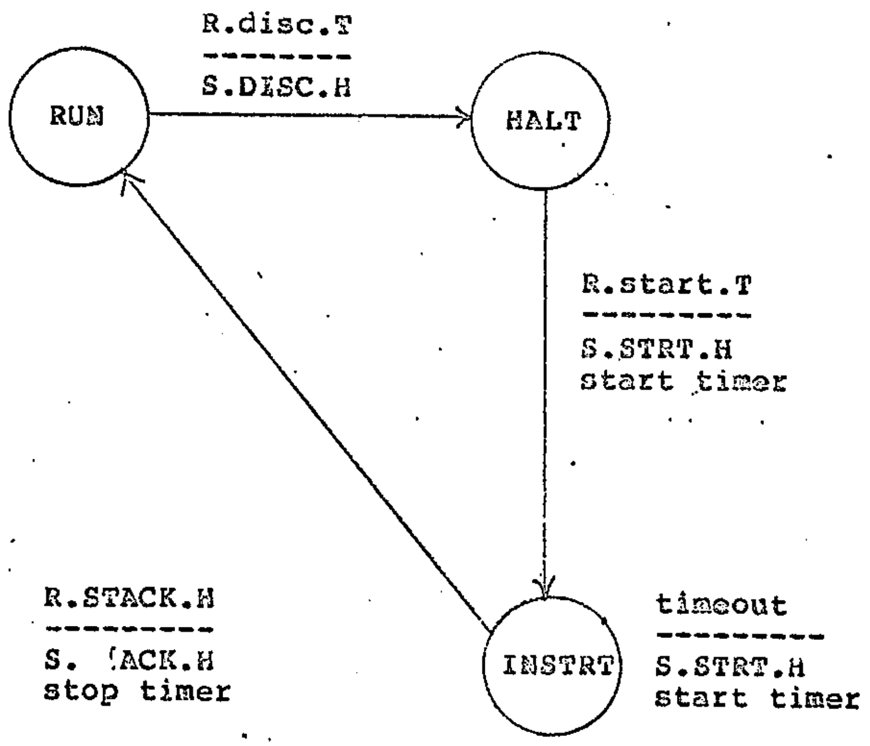
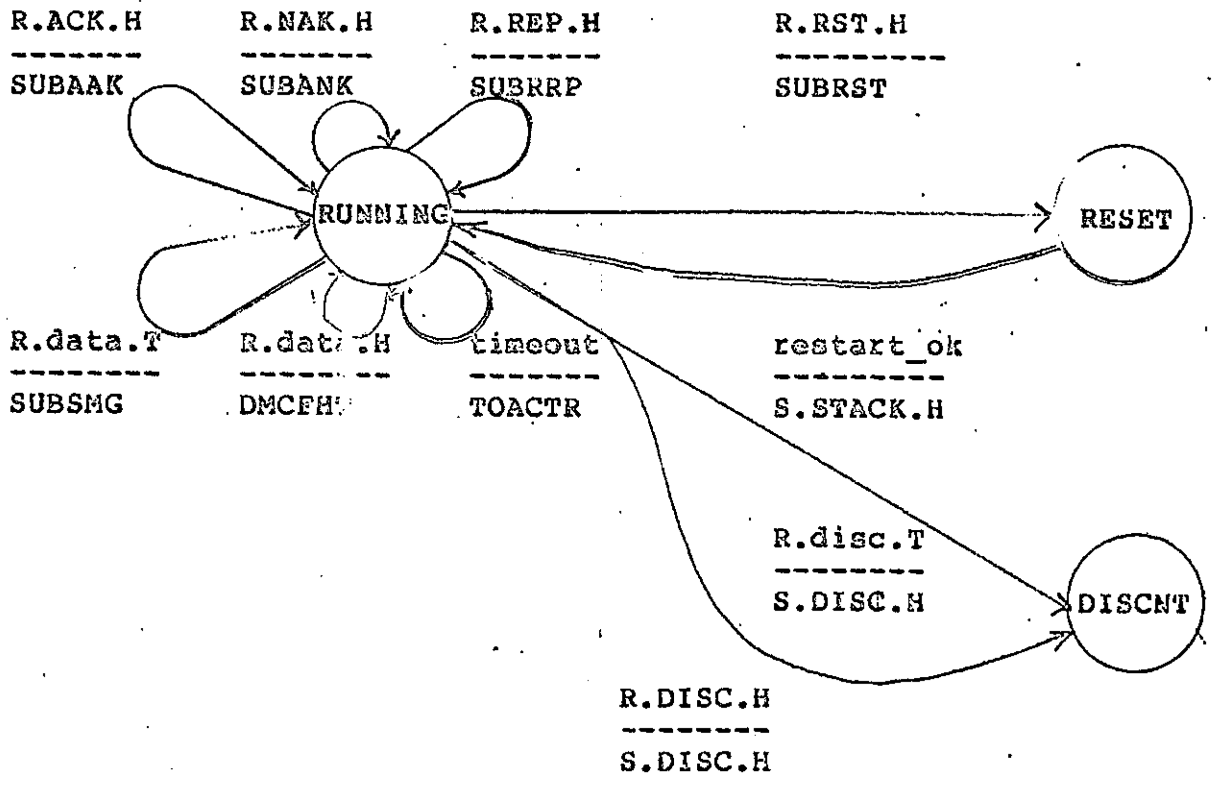


Figure 4.4 DLC connection protocol



R.	:	receive
S.	:	send
.T	:	terminal
.H	:	host

Figure 4.5 Data exchange protocol of DLC

The software implementation of the DDCMPH program employs the table-driven technique. The merit of this technique is simple, flexible, and extensible.

With table-driven technique, the main program continuously polls the occurrence of events(an event may be "information frame received", "supervision frame received", " timeout", etc.) and executes the required actions by examining the events table. The program is always existing in a 'state' which represents the current situation of the communicating entity. Whenever an event occurs, the corresponding actions will be taken, and the program is changed into another state. Figure 4.6 illustrates the relationship between state, event, and action.

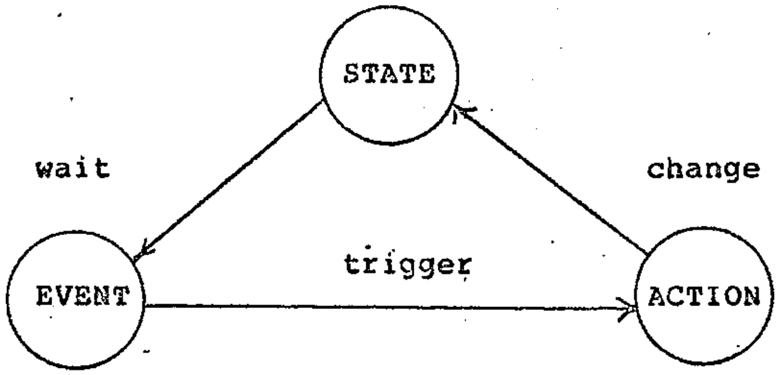


Figure 4.6 Relationships between state, event, and action

Table 4.1 shows the states, events, and actions of the DDCMPH program.

current state	event	action state	next
RUNNING	R.ACK.H	SUBAAK	RUNNING
RUNNING	R.NAK.H	SUBANK	RUNNING
RUNNING	R.REP.H	SUBRRP	RUNNING
RUNNING	R.data.T	SUBSMG	RUNNING
RUNNING	R.data.H	DMCFHT	RUNNING
RUNNING	timeout	TOACTR	RUNNING
RUNNING	R.RST.H	SUBRST	RESET
RUNNING	R.disc.T	S.DISC.H	DISCNT
RUNNING	R.DISC.H	S.DISC.H	DISCNT
RESET	restart_ok	S.STACK.H	RUNNING

Table 4.1 State table

The function of each action routine listed in Table 4.1 is described below.

.SUBSMG -- Process and send a data frame to host computer.

* format frame data by adding frame header.
* send data frame to Host.

.DMCFHT -- Process incoming information frame.

* check sequence number of the data frame.
* send ACK or NAK to Host.
* move the data to user via SWCCB.

.SUBAAK -- Process ACK frame.

* check R# in received frame, if match, update the R# in the buffer.

.SUBANK -- Process NAK frame.

* check R# in received frame, if match, send data frame once more.

.SUBRRP -- Process REP frame.

* check the reply sequence number,
  if match, send ACK to Host,
  else send NAK to Host.

.SUBDIC --Disconnect FEP and Host.

* send a DISC frame to Host.

.TOACTR -- Response to timeout.

* check count of timeout, if over, send RESET frame to Host, else send REP frame to Host.

.SUBRST -- Reset protocol machine.

* reinitialize running conition.
* restart.

4.3 UTERMH -- User Terminal Handler

The main functions of the UTERMH task are:

- Handle user login procedure.
- Receive the user's key-in data and send it to the task in the host computer via DDCMPH, and vice versa.

UTERMH task makes the problem of line concentration be transparent to the user. To the terminal user, a terminal connected to the host computer via FEP has no difference with the terminal directly connected to the host computer.

UTERMH task communicates with the peer-protocol task in the RSX-11/M system by following a high-level communication protocol. This control protocol is illustrated with a state diagram in Figure 4.7. At running state, FEP multiplexes the user's data to/from the host computer. This multiplexing is completely transparent to the user.

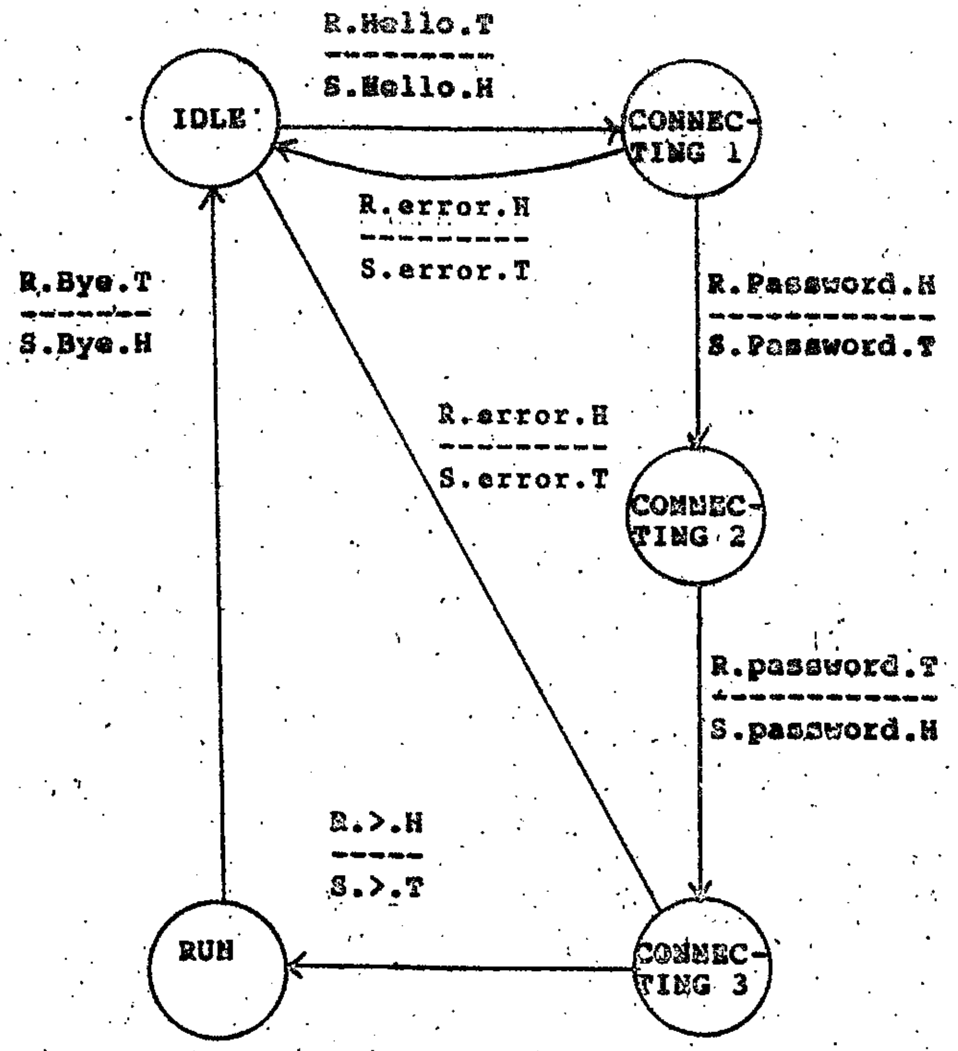


Figure 4.7 Remote Login Protocol

In FEP, when the UTERMH task has received a line data from the user, it will tailer the data into a message by adding a control header(see Figure 4.8) which is to be examined by its peer level protocol handler in RSX-11/M system, and then pass the message to DDCMPH task via a software channel control block. The DDCMPH task will treat the whole message as data and format it as we described in Section 4.2.

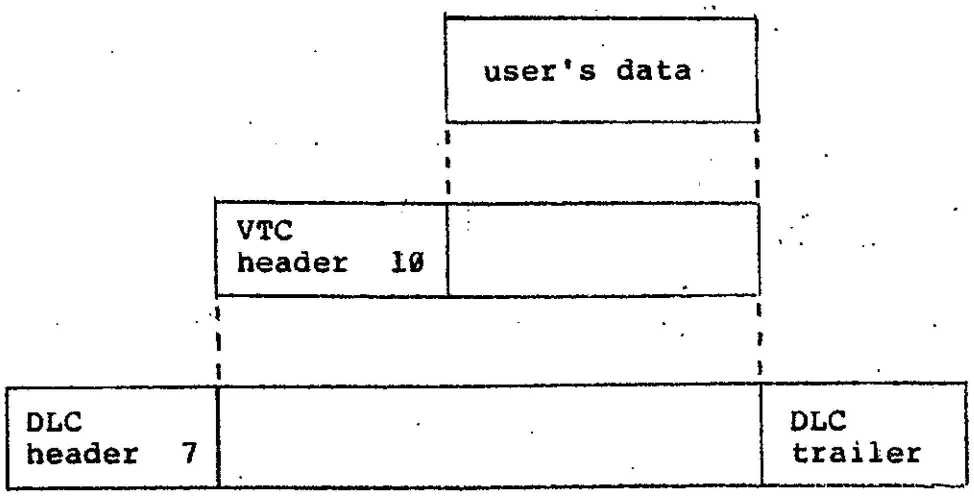


Figure 4.8 Onion structure of message frames

5. RSX-11/M Network Communication Software

This section describes the implementation detail of the communication software in RSX-11/M operating system. The protocol hierarchy of this implementation is shown in Figure 5.1. To achieve the functional capability of this protocol hierarchy, we have modified the original RSX-11/M to enhance its communication capability. Essentially, the mechanism we employed is centralized on the concept of virtual terminal, through which multiplexing technique can be achieved.

5.1. Virtual Terminal Concept

In RSX-11/M, each job (a log-in user) is usually initiated by a user at a physical terminal. Many tasks may run concurrently with input and output through their associated terminal devices. Each physical terminal is associated with a control block of memory in the operating system, which contains information about the physical terminal and I/O request buffers as the linkage between the physical terminal(user) and the job(running task).

In computer communication software, it is desirable to allow a task in the system to be initiated and controlled by another task, usually the communication control task, instead of a user at a physical terminal. All I/O data of the controlled task should be passed through the controlling task. The controlling task cannot use a physical terminal in the usual way, some means must be provided in the executive for the controlling task to send input to and accept output from the job being controlled.

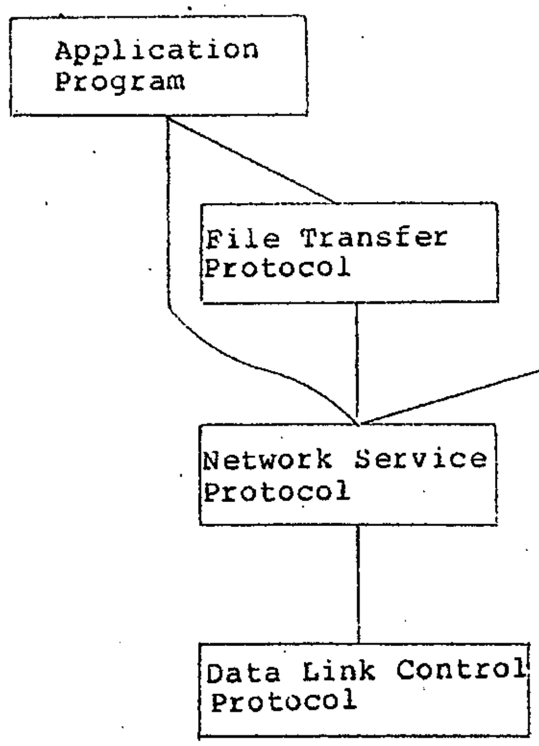


Figure 5.1 Layered Protocol Structure in RSX-11

Unfortunately this facility is not provided by the original RSX-11/M operating system. Hence, we introduce a pseudo device, said virtual terminal(VT), to provide this capability. The virtual terminal is a simulated terminal and is not defined by the hardware. Like hardware-oriented terminals, each VT has a control block of memory associated with it. This block of memory is used by the VT in the same manner as a hardware-oriented terminal uses its control block memory. Figure 5.2 shows the parallelism between a hardware-oriented physical terminal and a software-oriented virtual terminal.

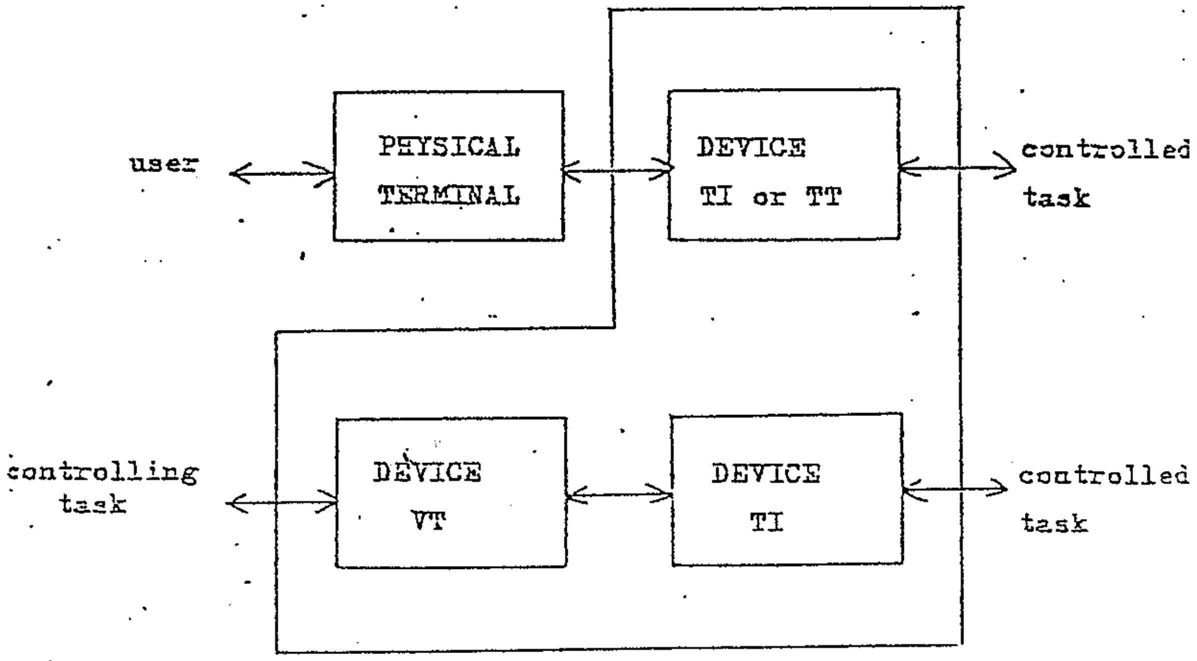


Figure 5.2 Parallelism between PT and VT

The controlling task uses the VT in the same way as the user uses a physical terminal device. It initiates the VT, input characters to and wait for output from the VT, and closes the VT using the appropriate programmed facility. The controlled task performs I/O to the VT as though the VT were a physical terminal.

The controlled task may get into a loop and not accept any input from its associated VT device, therefore, it is not possible for the controlling task to simply rely on busy-waiting for events or activities of the controlled task. The controlling task may wish to drive more than one controlled task, and be able to respond to any of these tasks; therefore, the controlling task cannot stationarily wait for any particular VT. For these two reasons, the VT differs from other devices in that it is never in an I/O wait state. Synchronization between the controlling task and the controlled task is accomplished by event flags and asynchronous system traps(AST) provided by the RSX-11M operating system.

Event flags are a means by which tasks recognize specific events. In requesting a system operation such as an I/O transfer, a task may associate an event flag with the desired I/O operation. When the I/O complete event occurs, the executive will set the specified flag. Each event flag has a corresponding unique Event Flag Number (EFN). A task can set, clear, and test event flags to check whether the specific event occurs or not. A task may also wait on more than one event flag to monitor many outstanding asynchronous activities.

Asynchronous System Traps(AST) detect events that occur asynchronously to the task's execution. That is, a task has no direct control over the precise time that the event(the trap) may occur. For example, the completion of an I/O transfer may cause an AST to occur. The primary purpose of an AST is to inform the task that a certain event has occurred. As soon as the task has serviced the event, it can return to the interrupted code. ASTs can be used as an alternative to event flags or the two can be used together. Users can specify the same AST routine for several external activities, each with a different event flag. When the executive passes control to the AST routine, the event flag can determine the action required.

The controlling task in the host may create several VT channels for the remote users. The same input and output AST routines are specified for every VT device. When an I/O request is issued from the controlled task, the AST routine will be entered. In AST routine, it checks which device the I/O request comes from and sets a specific event flag associated to that channel. Thus, the main program can monitor all the event flags to handle I/O conditions of VT devices. The synchronization between the controlling task and the controlled tasks is achieved by this mechanism.

5.2. Virtual Terminal Device

The RSX-11M I/O system is structured as a hierarchy in Figure 5.3. At the top of the hierarchy are file control service(FCS) and record management services(RMS), which provide device-independent access to devices included in the system. The QIO directive is the lowest level of task I/O operations. The QIO directive allows direct control over devices that are connected to a system and that has an I/O driver. The I/O services provided by the executive consist of QIO directive processing, and a collection of subroutines used by drivers to obtain I/O requests, and facilitate interrupt handling. The actual control of the device is performed by the driver.

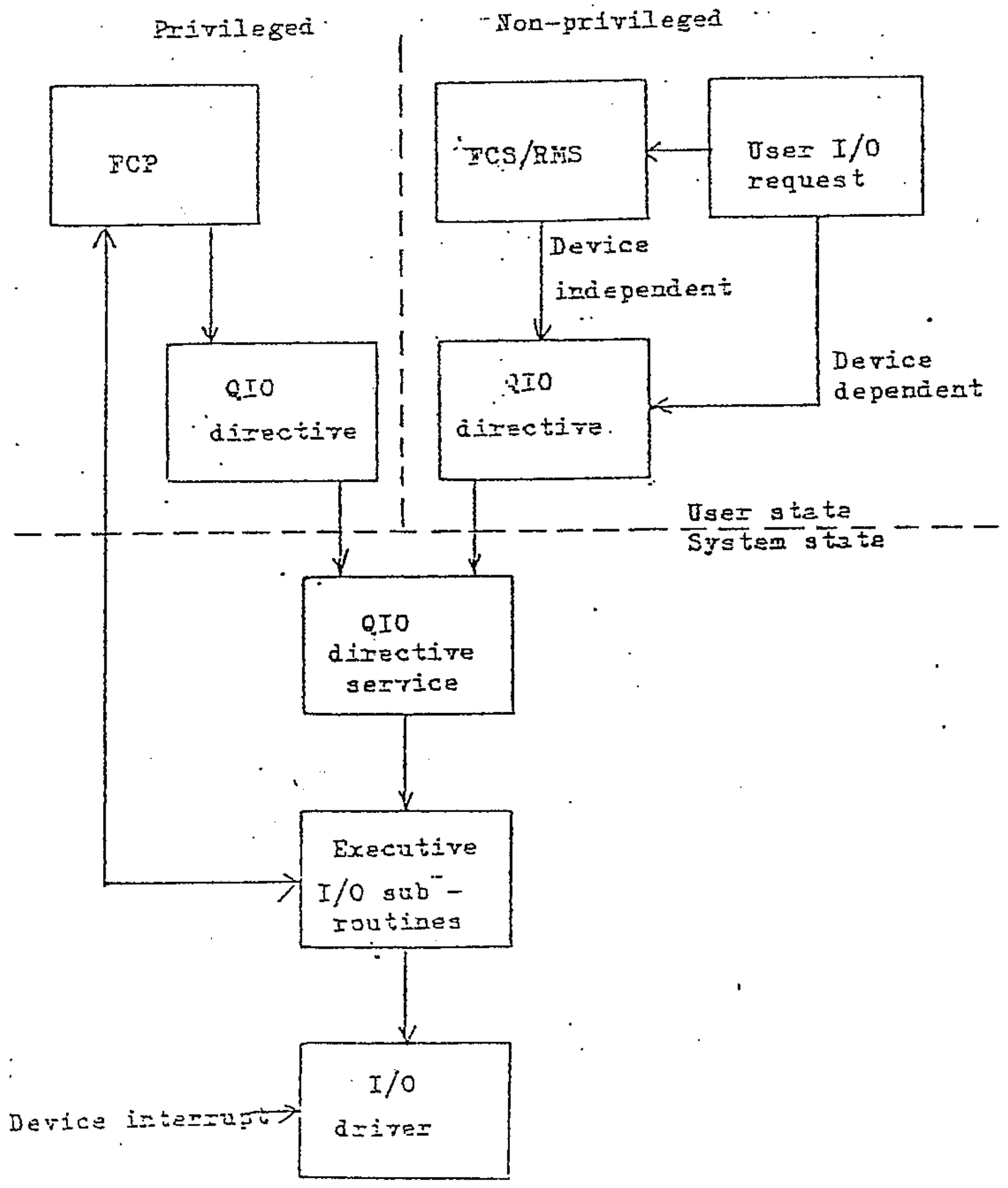


Figure 5.3 RSX-11/M I/O flow control

There are four data structures important to the driver; all I/O operations are controlled by these data structures. They are :

(1) the Device Control Block(DCB).

(2) the Unit Control Block(UCB).

(3) the Status Control Block(SCB).

(4) the I/O packet.

(5) AST Block.

Figure 5.4 through Figure 5.8 illustrate the contents of these data structures. For detail of these data structures, the reader should refer to [7]. Once a device has been created, the related data structures must be established. When a VT channel is opened, the data structures for a VT unit Device Control Block (DCB) and Unit Control Block(UCB) are created and linked into the device list and assigned with the lowest available VT unit number. Three AST routines may be specified, the input AST, the output AST, and the attach and detach AST.

The controlling task(VTMON) can service each offspring(controlled) task's input or output request with a corresponding output or input request to the correct virtual terminal unit. For example, suppose that a controlled task has been activated as an offspring task of the remote log-in control task and associated with VT2: as TI:, then

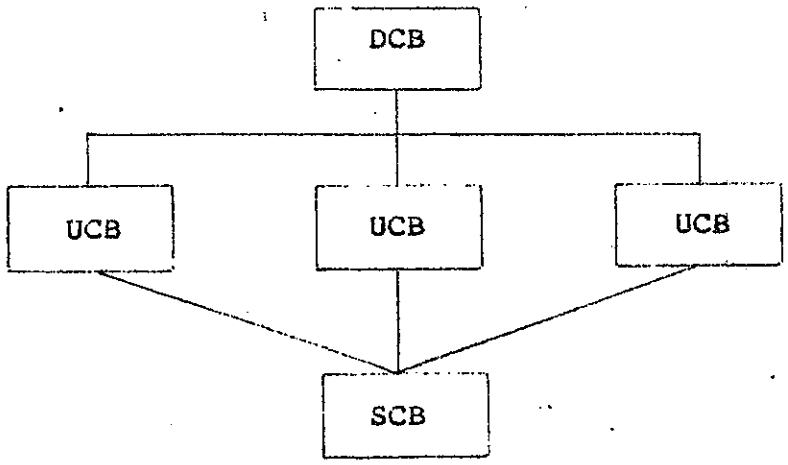
1. Offspring issues an IO.RVB or IO.RLB to TI: for its input line. The virtual terminal driver queues the request internally and effects an AST in the remote log-in control task

D.LNK	Link to next DCB (0=last)		0
D.UCB	Link to first UCB		2
D.NAM	Generic device name		4
D.UNIT	Highest unit no.	Lowest unit no.	6
D.UCBL	Length of UCB		10
D.DSP	Address of driver dispatch table		12
D.MSK	Legal function mask bits 0 - 15.		14
	Control function mask bits 0 - 15.		16
	No-op'ed function mask bits 0 - 15.		20
	ACP function mask bits 0 - 15.		22
	Legal function mask bits 16. - 31.		24
	Control function mask bits 16. - 31.		26
	No-op'ed function mask bits 16. - 31.		30
	ACP function mask bits 16. - 31.		32
D.PCB	Address of partition control block		34

Figure 5.4 Device Control Block

U.LUIC	Log-on UIC		–4
U.OWN	Owning terminal UCB address		–2
U.DCB	Back pointer to DCB		0
U.RED	Redirect UCB pointer		2
U.CTL U.STS	Unit status	Control flags	4
U.UNIT U.ST2	Unit status	Physical unit no.	6
U.CW1	Characteristics word 1		10
U.CW2	Characteristics word 2		12
U.CW3	Characteristics word 3		14
U.CW4	Characteristics word 4		16
U.SCB	Pointer to SCB		20
U.ATT	TCB address of attached task		22
U.PKT	Address of I/O packet		24
U.PTCB	TCB address		26
U.IAST	Input AST address		30
UOAST	Output AST address		32
UAAST	Attach AST address		
	• • •		
	storage		

Figure 5.5 Unit Control Block



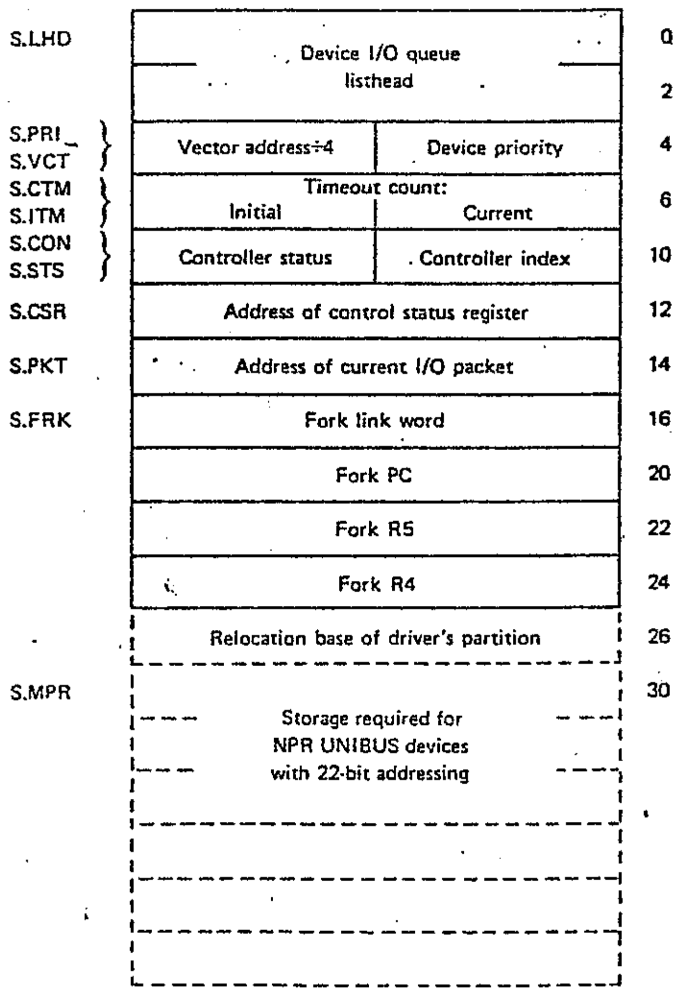


Figure 5.6 Status Control Block

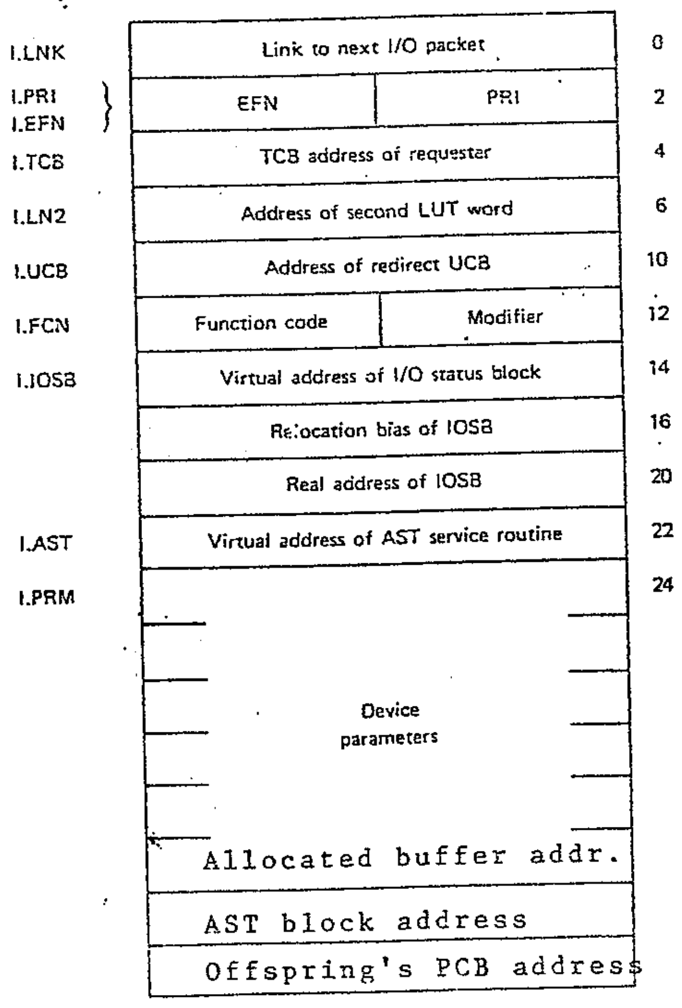
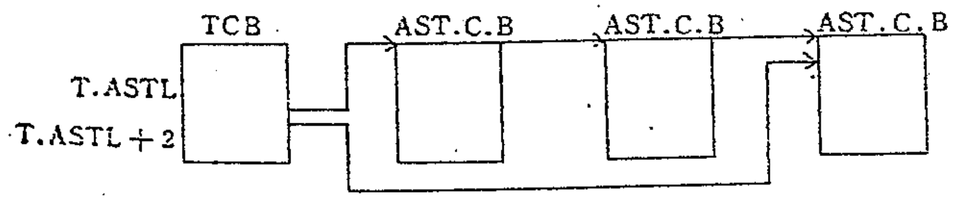


Figure 5.7 I/O Packet Format



A.KSR5(-4)	Subroutine KISAR5 Bias (A.CBL=0)
A.DQSR(-2)	Dequeue Subroutine Address (A.CBL=0)
0	AST Queue Thread Word
A.CBL	Length of Control Block in BYtes
A.BYT	Number of Bytes to Allocate on Task Stack
A.AST	AST Trap Address
A.NPR	Number of AST Parameters
A.PRM	First AST Parameters

Figure 5.8 AST block

at the virtual address "IAST" with the unit number 2 and the byte count from offspring's I/O request on the stack.

2. In the controlling task's AST routine, an event flag associated with that channel is declared. The remote log-in control task detects this event, retrieves an input line for offspring from the physical I/O port, and specifies this line in a QIO directive to a LUN assigned to VT2: with an IO.WVB or IO. WLB.

3. The virtual terminal driver reads the line from the control task's buffer, writes the line to offspring's buffer and then signal I/O completion for both I/O requests. Similarly, if offspring needs to print a message, it does so with an IO.WVB or IO.WLB to TI:.

4. In the controlling task's output AST routine, a specfic event flag is also set. For the declared event flag, the control task issues an IO.RVB or IO.RLB to retrieve the line via the virtual terminal driver. Then, the control task may output this line to the physical I/O port with the user ID in front of the message.

For each remote user's command line the control task may use spawn directive queuing a command line to a specific task for execution, and establishing the task's TI: as a previously opened virtual terminal unit. The task being spawned is a command line interpreter called Monitor Console Routine(MCR), it allows users to operate and control the RSX-11M system. The I/O operation process of the virtual terminal device has been illustrated in Figure 5.9. Figure 5.10 shows the control flow of VTMON.

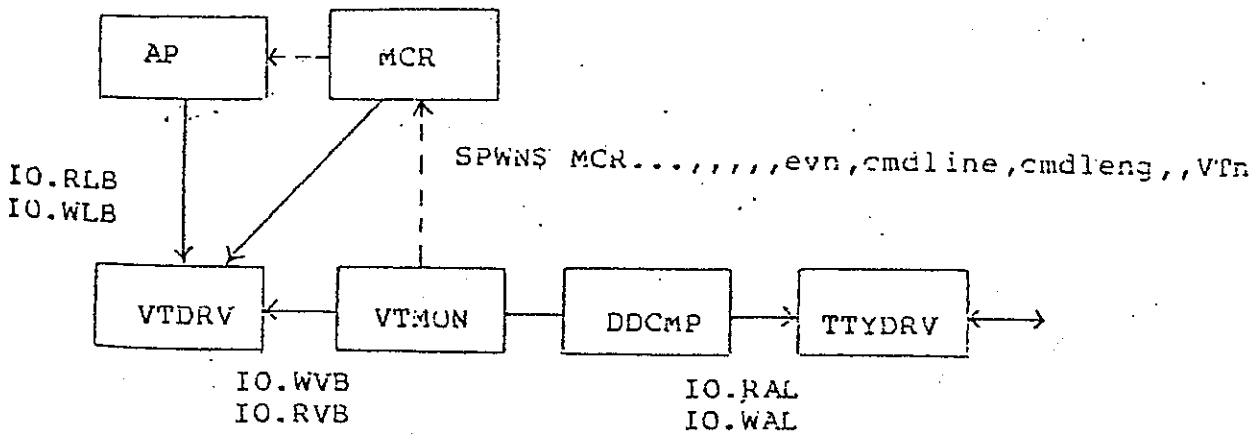


Figure 5.9 Modified RSX-11/M Terminal I/O

```
VTMON:
        initialization;
   MAINLP:
        wait events;
        event 10: /* data from DDCMP */
           pack input data, set flag 9 if got whole frame;
           goto MAINLP;
        event 9 : /* process frame data */
           get frame data from the ring buffer;
           check user ID;
           if VT is not created yet then
                create new VT by linking DCB and UCB,
                spawn command(Hello) to MCR;
                goto MAINLP;
              else if input request has been set then
                        issue $QIO IO.WLB to VTDRV,
                        else spawn command to MCR;
              goto MAINLP;
        event 1-8: /* I/O request from VTDRV */
           scan event flag list and store the right VT# in R1;
           if it's input request from VTDRV then
                set input request flag;
                clear event flag;
                goto MAINLP;
              else
                clear event flag;
                issue $QIO IO.RLB to VTDRV;
                check VT status, if it's illegal hello message then
                                                  close VT;
                                              else if it's bye message then
                                                  set MARKTIME;
                transform VFC(in data buffer) into characters;
                format data frame by adding header;
                send output buffer to DDCMP;
                goto MAINLP;
```

Figure 5.10 Control flow of VTMON

5.3. Virtual Terminal Driver

The virtual terminal dirver is primar ly intended for facilitating a parent task to simulate terminal I/O for an offspring task activated with the spawn directive. This simulation takes place via a virtual terminal unit whose unique data structures(DCB and UCB) are dynamically created while the virtual terminal unit is being opened. Only one common SCB is used for all virtual terminal units.

The virtual terminal driver employs the UC.QUE bit(in UCB U.CTL word) to receive all I/O packets directly from the QIO directive. Offspring read- and write- requests are queued to the common SCB and dequeued one by one (FIFO), based on the attachment of the device and the presence of other requests. Whenever an offspring read or write is dequeued, the parent task receives an AST at its input- or output- AST entry point. The parent task is then expected to issue a complementary write- or read- request to simulate a terminal I/O transfer.

Only Offspring tasks may attach the virtual terminal unit. Parent task's requests are always serviced in spite of the attachment of the virtual terminal unit.

The driver initiator entry point is entered from the QIO directive whenever a parent or offspring request is issued. Parent I/O requests are always serviced immediately, normally resulting in a block I/O transfer of data and the completion of both the parent request and the corresponding offspring request.

Offspring requests are initially queued and then dequeued one by one. An AST is declared in the parent task whenever an offspring read or write is dequeued. Figure 5.11 shows the control flow of the VT driver.

```
VTDRV:
   Check U.PTCB, if not VTMON requests I/O then goto OFSRIO,
        else /* VTMON requests I/O */
           transfer data;
           if this is offspring task's write request then
                deallocate buffer;
           set unit not-bust(U.STS);
           if it's buffered I/O then
                queue I/O packet in AST list of offspring task's TCB,
                unstop offspring task;
           do I/O finish(via $IOFIN) for offspring task;
                /* $IOFIN will deallocate I/O packet and
                   send I/O complete status */
           do I/O finish(via $IOFIN) for VTMON;
           goto GIOPKT;
        OFSRIO: /* offspring task requests I/O */
           queue I/O packet in SCB;
        GIOPKT: get I/O packet from SCB;
           if no more request or unit busy then return,
                else /* process offspring task's I/O request */
                   set unit busy(U.STS),
                   save address of I/O packet in U.PKT,
                   allocate AST block,
                   save address of AST block in I/O packet;
                   if not buffered I/O then goto QUEAST,
                        else /* buffered I/O */
                           store PCB address in I/O packet,
                           allocate buffer and save address
                                in I/O packet;
                           if read request then goto STOPOF,
                                else transfer data from the
                                        user's buffer to
                                        allocated buffer;
                           STOPOF: stop the offspring task;
                   QUEAST: queue AST block in TCB of VTMON;
return;
```

Figure 5.11 Control flow of VTDRV

5.4 DDCMP -- Data Link Control Protocol

The data link control protocol in RSX-11/M system is completely symmetric to the data link control protocol in FEP. However, since that we are also working for a project to connect a heterogeneous computer network for a distributed database system[14], the DDCMP protocol handler in the RSX-11/M system has been implemented to couple with the network's requirements. Presumably the link connection protocol has been extended as balance mode in the sense that either of the two stations can initiate the link connection, this is different from the master-slave mode of the FEP's link connection protocol. Figure 5.12 illustrates the DLC connection protocol of the RSX-11/M system. The DLC data exchange protocol is the same as shown in Figure 4.5.

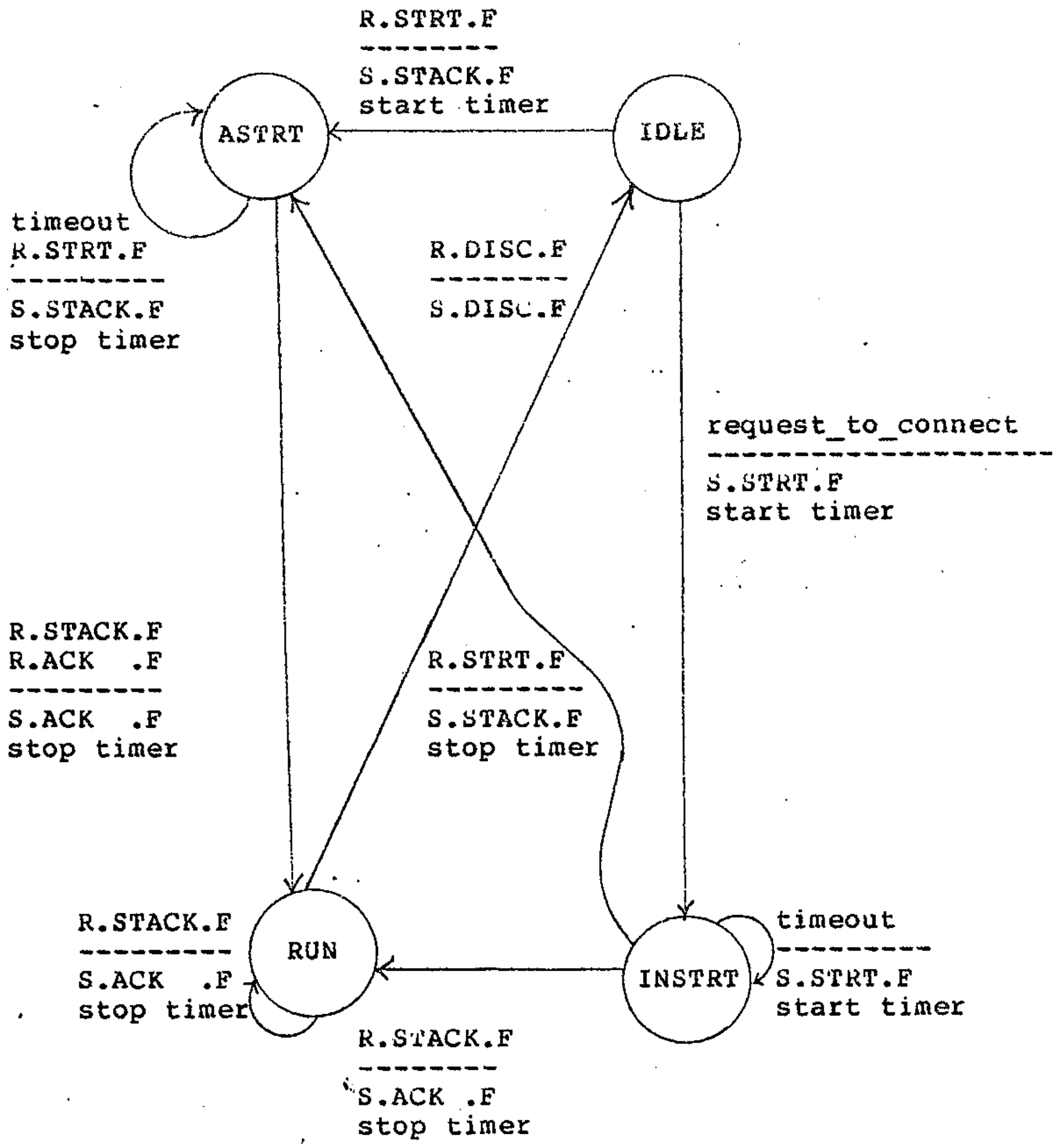


Figure 5.12 DLC connection protocol of RSX-11/M

Conclusions

In this report we describe the design and implementation detail of a Z80-based front end processor with multiuser remote login capability. In FEP, a real-time event-driven multitasking executive has been implemented to drive a set of layered protocol machines. In the host computer, a set of comunication software programs has also been implemented to function as the layered peer-protocol machines. In particular, virtual terminal concept has been employed to enhance the communication capability of the RSX-11/M system. The result of this project has also been extended to achieve the communication requirement of a heterogeneous computer network, and is now run to test.

REFERENCES

(1) D. M. Ritchie and K. L. Thompson,"The UNIX Time-sharing System," CACM, VOL.17, NO.7, July 1974.

(2) R. F. Rashid,"An Interprocess Communication Facility For UNIX," Local Networks for Computer Communications, North-Holland Publishing Company, 1981.

(3) L. A. Rowe and K. P. Birman,"A Local Network Based on the UNIX Operating System," IEEE Trans. on Software Engineering, VOL. SE-8, NO.2, March 1982.

(4) "RMX-8∅ User's Guide", Intel Corp.

(5) "REX-8∅ Primer", System and Software Corp.

(6) "RSX-11/M Utility Manual", Digital Equipment Corp.

(7) "RSX-11/M Executive Reference Manual", Digital Equipment Corp.

(8) D.W.Davies "Computer networks and their protocols"

(9) Michel Gien "A File Transfer Protocol",Computer Network, Feb. 1978.

(1∅) R. C. Holt, "Structured Concurrent Programming with Operating System Applications," Addison Wesley, 1978.

(11) John D.Day,"Terminal Protocols" IEEE transaction on communications, VOL.COM-28, no.4, APRIL 198∅.

(12) Kai Hwang, Benjaming W.Wah and Fage A.Briggs, "Engineering Computer Network",AFIPS conference proceedings VOL.5∅.

(13) J. S. Ke and L. C. Liu,"Design and Implementation of a Front End Processor with Multiuser Editing

Capability", Technical Report TR-81-ØØ4, Institute of Information Science, Academia Sinica, R.O.C., October 1981.

(14) J. S. Ke, C. L. Lin, and H. L. Chen etal.,"Design and Implementation of a Distributed Database System", Technical Report TR-83-ØØ2, Institute of Information Science, Academia Sinica, R.O.C., January 1983.

(15) C. C. Lu, C. L. Lin, and J. S. Ke,"Enhanced Interprocess Communication Mechanisms for UNIX", Proc. of ICS82, Taichung, Dec. 1982.