MIT/LCS/TM-11

DESCRIPTION AND FLOW CHART OF THE

PDP-7/9 COMMUNICATIONS PACKAGE

Philip W. Ward

July 1970
Unnumbered Blank Page

Note: This page is an unnumbered blank used to preserve the printed appearance of the author's original document. It is the back side of the previous numbered page.
ACKNOWLEDGMENT

Work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01).
Note: This page is an unnumbered blank used to preserve the printed appearance of the author's original document. It is the back side of the previous numbered page.
DESCRIPTION AND FLOWCHART YIELD DESCRIPTION OF THE LOGO 8/58 ENGLAND

PDF-7/8 COMMUNICATIONS PACKAGE

PROJECT MAC

Massachusetts Institute of Technology

Cambridge

Massachusetts 02139
The PDP-7/9 Communications Package was written to provide data transfers between the buffer controller (PDP-7 or PDP-9) of an ESL Display Console and a host computer via a 50-kilobit serial Dataphone link. Initially, only one of the displays (with a PDP-9 buffer controller) was to be operated remotely over a 50-kilobit line, and the only feasible access to the 7094 CTSS host computer was via the PDP-7 buffer controller of the other display, which is directly connected to CTSS channel D. For this connection, the PDP-7 could be looked upon as the "host" for the PDP-9, although it merely served as a message-handling intermediary for the real host, the 7094.

The link between the PDP-9 located at Project MAC (Technology Square) and the PDP-7 located at the M.I.T. Information Processing Center was installed in May, 1969. The communications package described herein was successfully checked out, but integration with the display executive programs of the PDP-7 and PDP-9 to permit remote display operation had not been accomplished when work was terminated in March, 1970.

The work described was performed by the Display Group of the M.I.T. Electronic Systems Laboratory, with the joint support of Project MAC and the U.S. Air Force Materials laboratory, Wright-Patterson AFB, under Contract F 33615-69-C-1341. The programs were written and debugged by D.E. Thornhill, H. Levin, and M.F. Brescia. This description by P.W. Ward was prepared as a user's guide.

John E. Ward
Electronic Systems Laboratory
## CONTENTS

**DESCRIPTION OF THE PDP-7/PDP-9 COMMUNICATIONS PACKAGE**
- Introduction ........................................ 1
- Overview of Communications Package .......... 1

**CONVENTIONS FOR CALLING PDP-7/PDP-9 COMMUNICATIONS PACKAGE**
- Initialization ....................................... 7
- Receive Set ......................................... 8
- Scheduling Routine ............................... 9
- Receive Message ................................... 11
- Send Message ....................................... 14
- Send Message Master/ Slave Modification ..... 17

**SUGGESTED PROGRAM MODIFICATIONS** ........... 18

**Appendix I -- Flow Chart of Communications Package** ............... 20

**Appendix II -- Communications Package Error Parameters** .......... 52
Note: This page is an unnumbered blank used to preserve the printed appearance of the author's original document. It is the back side of the previous numbered page.
DESCRIPTION OF PDP-7/PDP-9 COMMUNICATIONS PACKAGE

Introduction

The program to be described was written specifically for the purpose of providing a message handling facility between a PDP-7 and a PDP-9 computer utilizing a 50 kilobit telephone transmission link (see Figure 1). Each computer is physically connected to the telephone media (typically a Bell 303 Modem) via a DEC 637 Interface*. The 637 conforms (at the modem interface) to the Electronic Industries Association Standard RS-232-B for full duplex operation. At the 637-to-modem level, information is transmitted and received in serial bit synchronous form. (In addition, the Bell Modem "scrambles" and "descrambles" the bit stream to provide uniform spectrum distribution and utilization.) At the Computer-to-637 Interface level, information is transmitted in serial byte synchronous form. In this program implementation, one byte is an 8-bit character; but 6, 7 or 9 bit options are possible with the same 637 Interface unit. The message handling program communicates with the 637 by Input-Output-Transfer (IOT) Commands which provide the status and control information required to effect transmission (and reception) of 8 bit characters from (and to) the PDP-7/9 Accumulator.

Overview of Communications Package

The user of the Communications Package interacts with essentially three subprograms (hereafter called procedures):

*The 637 Interface is also referred to by Digital Equipment Corporation as "Bit Synchronous Data Communication System Type 637" or simply "637 Data Communication Channel."
FIGURE 1 - BLOCK DIAGRAM OF PDP-7/9

COMMUNICATIONS LINK
(1) Initialization

(2) Receive Message

(3) Send Message

How the user calls these procedures will be treated individually following a brief overview of what the procedures do.

Figures 2, 3 and 4 illustrate in block form the general flow of operations within a given procedure. For a detailed Flow Chart of the Communications Package refer to Appendix I.

Referring to Figure 2, the Initialization procedure clears the parameter list used by the program, puts the modem into synchronization, waits for an indication that the remote user has initialized, then returns control to the user.

Figure 3 illustrates the Receive Message procedure. Note that the user, in general, interacts with this procedure at three different instances.

(1) User calls Receive Set to provide parameters needed to process an incoming message.

(2) User calls Receive Message when he knows a message is forthcoming.

(3) During the procedure (2), if the entire message is received successfully, the procedure calls the user’s scheduling routine. This step not only provides the user with an indication that there were no transmission errors, but also a chance to call procedure (1) again. Thus, an appropriate scheduling routine will prevent overwriting of a message by a subsequent message. After the scheduling routine is complete, control is returned to procedure (2) for completion.

The Send Message procedure is illustrated in Figure 4. In this case, the user invokes the procedure and passes the necessary parameters at the same instance. The procedure attempts to send the message, and, if successful,
**FIGURE 2 -**

BLOCK DIAGRAM OF
INITIALIZATION PROCEDURE

**Finit: Entry**

- **Zero Communications Package Parameters**

- **Initialize 637 Interface**

- **Synchronize Modem Transmit Link**

- **Wait for Indication of Modem Receive Link Synchronization**

- **Return**

- **Timeout Error**

- **Halt**
Figure 3 -

Block diagram of receive message procedure.
```
SNDMES: ENTRY 6

GET 5 PARAMETERS:
1) USER NUMBER SENDING TO:
2) USER NUMBER SENDING FROM
3) SEND MESSAGE LOCATION
4) LENGTH OF MESSAGE
5) FORMAT: ASCII OR BINARY

SET UP POINTER TO DIRECT ANY TIME OUT ERROR TO "SNDTIME"

SEND "SYN" TO PLACE RECEIVER IN RECEIVE ACTIVE
SEND "ENQ" AS REQUEST TO SEND MESSAGE

WHEN "ACK" IS RECEIVED
SET UP TO SEND MESSAGE
ELSE, ERROR RETURN

SEND MESSAGE HEADER
MAINTAIN CHECKSUM
SEND MESSAGE TEXT
SEND "ETX", CHECKSUM

SEND "NULL" IN IDLE MODE
CONTINUE IF "ACK" RECEIVED
TRY AGAIN IF BAD CHECKSUM
ELSE, ERROR RETURN

SEND "EOT" IN IDLE MODE
CLEAR RECEIVE ACTIVE,
RECEIVE ONLY "SYN"

RETURN
ERROR RETURN
```

FIGURE 4 -

BLOCK DIAGRAM
OF SEND MESSAGE
PROCEDURE
returns control to the user at his normal return entry point. Otherwise, control is returned to the user's error return.

The program takes care of all of the overhead and the input/output operations required with the 637 Interface to get a message processed. The user may opt that the message text to be sent consist of ASCII (non-control) characters or 18-bit binary words. The message header, checksum, send/receive protocol, and the assertion of a program interrupt before sending a message, come under the overhead items which are taken care of by the program. However, the burden is on the user to:

(1) Provide the required interrupt service for receiving a message or, alternatively, if the priority interrupt facility has been disabled, a routine to determine the presence of a message.

(2) Inhibit interrupts when sending a message.

(3) Recover under error return conditions.

CONVENTIONS FOR CALLING PDP-7/PDP-9 COMMUNICATIONS PACKAGE

Now that some insight has been presented concerning what the program does, the specific details of invoking the procedures in the program will be treated in the same order.

Initialization

The first procedure that the user will invoke is initialization. This is accomplished by calling FINIT with no arguments. A typical call follows:

```
JMS FINIT /CALL FINIT WITH NO ARGUMENTS
```

ARLE,
After initialization is accomplished, control will return to the instruction at ABLE. If initialization cannot be accomplished the program will come to a halt and the operator must take appropriate corrective action, then start over. The initialization routine assumes someone on the other end is also trying to initialize and will wait until he does so before returning control. Initialization not only brings the modem link up to sync, it also zeros the procedure parameter list. The names and descriptions of the parameters used to indicate error conditions are given in Appendix II. The user may wish to use the parameter list to determine his program action in case of error returns from the Receive Message or Send Message procedures.

Receive Set

Before the user invokes the receive message routine the first time, he must call RCVSET with three arguments. A typical call procedure follows:

```
  JMS RCVSET
  LAC ARG1R
  LAC ARG2R
  LAW ARG3R
  BAKER, .
```

Control is returned to the instruction at BAKER when RCVSET has accepted the three arguments. If a message should come in before RCVSET has been invoked by the user, it will be refused on the basis that no buffer space is available. If one comes in afterwards, it will be placed in the buffer space last specified. ARG1R and ARG2R are self explanatory, except possibly it should be clarified that ARG2R is the octal number of contiguous locations available in the receive buffer regardless of whether the data type is ASCII
or Binary. ARG3R is the entry point to the user's receive scheduling routine.

This example also serves to illustrate how arguments are passed in the PDP-7/9. When the program executes the instruction JMS RCVSET, control is transferred to the instruction in the next location after RCVSET and the address of the instruction LAC ARG1R is placed in location RCVSET. By convention, arguments are passed by writing one instruction for each argument which, when executed, will place the argument in the Accumulator. Thus, LAC ARG1R puts the contents of ARG1R into the Accumulator while LAW ARG3R puts in the address of ARG3R. How these arguments are taken at the invoked procedure is illustrated in a later example. We can assume that after the arguments have been taken, the address contained in RCVSET will be BAKER. Thus, control is returned to BAKER by executing the instruction: JMP I RCVSET. Note that even if the return were to the instruction LAC ARG1R, no problem occurs in the program.

Scheduling Routine

The Receive Message procedure invokes the scheduling routine if a message has been received successfully. It does this before it acknowledges the message to the sender. As explained previously, this is the time to call RCVSET again if the user expects another message before he is finished with the present one. At the other end, the sending procedure will only wait about 1.5 milliseconds real time for a reply, so the scheduline routine cannot be too time consuming.

The scheduling routine can be as simple as the following example:

ARG3R, 0
JMP I ARG3R

/ENTRY POINT SCHEDULING ROUTINE
/RETURN IMMEDIATELY
The above routine simply returns control back to the receive message program and defers any action on the received message until later. However, the user may also wish to store the arguments being passed at this point of the program, namely RCVBUF (same as ARG1R, first location where the message was placed); RCVCNT (the length of buffer space used); TO (user number to whom message is sent); FROM (user number from whom message is sent). Assuming this information is needed, an example of an alternative routine which accepts the above arguments follows:

ARG3R, 0 /ENTRY POINT SCHEDULING ROUTINE

XCT I ARG3R DAC RMSBEG AOM ARG3R /PUT RCVBUF INTO RMSBEG
XCT I ARG3R DAC RSLTH AOM ARG3R /PUT RCVNT INTO RMLTH
XCT I ARG3R DAC RMSTO AOM ARG3R /PUT TO INTO RMSTO
XCT I ARG3R DAC RMSFRM AOM ARG3R /PUT FROM INTO RMSFRM

; /POSSIBLE CHANGE IN RCVSET

JMP I ARG3R /RETURN

This routine puts the four arguments into user's locations named RMSBEG, RMLTH, RMSTO, RMSFRM respectively.

This routine also serves to illustrate how arguments are accepted in the PDP-7/9. The entry point to the routine contains no instruction. When ARG3R is invoked by a JMS instruction, the location of the next instruction is stored at ARG3R and control is given to ARG3R + 1. Following the convention for accepting arguments, the scheduling routine issues an XCT instruction indirected through ARG3R to access the first argument. This places the first argument in the Accumulator. This is followed by an AOM instruction to increment the pointer in ARG3R to the next argument. The process continues until all arguments are taken, leaving the pointer in ARG3R at the return
entry point of the invoking procedure. This routine continues with some user defined algorithm that may decide to change the RCVSET parameters for the next message. Ultimately, control is returned by the instruction: JMP I ARG3R.

Receive Message

Assuming that the Receive Set procedure has been furnished with the necessary housekeeping parameters, the Receive Message procedure may be invoked at any time there is an indication of a message being sent. However, the user is almost certain to encounter a timeout error condition if he invokes the Receive Message procedure arbitrarily. The best arrangement is to direct the invocation on an interrupt basis, since this guarantees that the 637 Interface has been activated by a sender and a message is forthcoming. In order to clarify this point, the conventions followed by the Communications Package in this regard are described. The last step of any procedure orders the 637 Interface:

1. Transmit link to repeatedly transmit an "EOT" control character in order to maintain the Bell Modem synchronization. This leaves the 637 Interface transmit link "idling" and requires no further IOT operations, but a synchronous bit stream is maintained in the modem as required.

2. Receive link not to accept any more characters unless the "SYN" control character is detected. Under this status, the 637 receive link continually checks the serial bit stream for any sequence of bits that match the "SYN" character code. When a match is found, the receive link activates, sets its interrupt line active and begins assembling every 8 bits following "SYN" into characters.
In accordance with this convention, the Send Message procedure initially transmits the "SYN" character (three times) to alert the receiver. See Fig. 5.

It is possible to test for a message received status without the use of the interrupt scheme, but this requires IOT commands to the 637 Interface. A typical routine which waits on a message to arrive, then calls RCVMES with no arguments is as follows:

```
DZM ARG3R          /CLEAR ENTRY POINT ARG3R
SNE                /SKIP IF THERE IS LINE CONTROL
JMP RPRINT         /GO TO LOST LINE CONTROL PRINT
SRF                /SKIP IF 637 IS RECEIVE ACTIVE
JMP .-3            /KEEP CHECKING UNTIL ACTIVE
JMS RCVMES         /CALL RECEIVE MESSAGE
CHARLY, LAC ARG3R  /ARG3R USED AS MESSAGE FLAG
SNA                /IF ARG3R = 0, BAD MESSAGE
JMP RMSERC         /GO TO ERROR RECOVERY ROUTINE
JMP RMSPRC         /GO TO MESSAGE PROCESS ROUTINE
```

Control is returned to CHARLY after a message has been processed by RCVMES. It is assumed that subroutines RPRINT, RMSERC and RMSPRC exist in the user's program. If no problems were encountered by RCVMES, ARG3R (the user's scheduling routine) will have been accessed. For this reason ARG3R is used as a flag. If ARG3R is non-zero, the program flow is to RMSPRC where the user processes the message. The other subroutines are determined by the user's application. Typically, RPRINT might be a routine which notifies the operator that the modem has lost its line control.
**Figure 5 - Character Sequence of Typical Message**

**Typical Sender Sending Sequence**

- **BEGIN**
- **WAIT 10 MSEC FOR RESPONSE**
- **E E (IDLE TO REPEAT)**
- **T T (IDLE TO REPEAT)**

**Typical Sender Receiving Sequence**

- **(RECEIVE INACTIVE)**
- **S S A N N (IDLE)**

**NOTE:**

- **MESSAGE HEADER**
  - **MESSAGE NUMBER MOD 77g**
  - **MESSAGE "TO" MOD 77g**
  - **MESSAGE "FROM" MOD 77g**

- **MESSAGE TEXT**
  - **F ASCII TEXT FORMAT** OR **G BINARY TEXT FORMAT**
RMSERC could be quite involved in checking the receive message parameter list until the source of error is found and some action taken dictated by the type of error found.

Send Message

If the user desires to send a message he issues a call to SNDMES with five arguments. A typical call procedure follows:

```
.IOF /DISABLE INTERRUPT
JMS SNDMES /CALL SEND MESSAGE WITH 5 ARGUMENTS BELOW
LAC ARG1S /TO USER NO.
LAC ARG2S /FROM USER NO.
LAC ARG3S /FIRST LOCATION OF MESSAGE
LAC ARG4S /LENGTH OF MESSAGE BUFFER
LAC ARG5S /0=ASCII, ELSE BINARY
DOG, JMP SNDERT /SNDERT=ENTRY MY ERROR ROUTINE
EASY, ION /EASY=MY NORMAL RETURN ENTRY POINT
```

If the message is acknowledged by the receiver, the procedure returns control to the instruction at location EASY, otherwise the return is an error return to DOG which must transfer control to the user's send error routine at SNDERT.

Unless the recipient of the message needs the TO and FROM numbers contained in ARG1S and ARG2S, these arguments can be any arbitrary constant, including 0. In any case the procedure only sends the rightmost six bits of either argument. The next two arguments are self explanatory, since the procedure must know where to get the message and how long it is.
### Figure 6 - USASCII Code Table

| Bit Positions | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|---------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
|               | NUL | DLE | SP | 0 | @ | P | ` | p |
| 0 0 0 0 1      |     |     |    |   |   |   |   |   |
| 0 0 0 1 0      |     |     |    |   |   |   |   |   |
| 0 0 1 1 0      |     |     |    |   |   |   |   |   |
| 0 1 0 0 0      |     |     |    |   |   |   |   |   |
| 0 1 0 0 1      |     |     |    |   |   |   |   |   |
| 0 1 1 0 1      |     |     |    |   |   |   |   |   |
| 0 1 1 1 0      |     |     |    |   |   |   |   |   |
| 1 0 0 0 0      |     | BS  |    |   |   |   |   |   |
| 1 0 0 1 0      |     |     |    |   |   |   |   |   |
| 1 0 1 0 0      |     |     |    |   |   |   |   |   |
| 1 0 1 1 0      |     |     |    |   |   |   |   |   |
| 1 1 0 0 0      |     |     |    |   |   |   |   |   |
| 1 1 0 1 0      |     |     |    |   |   |   |   |   |
| 1 1 1 0 0      |     |     |    |   |   |   |   |   |
| 1 1 1 0 1      |     |     |    |   |   |   |   |   |
| 1 1 1 1 0      |     |     |    |   |   |   |   |   |
| 1 1 1 1 1      |     |     |    |   |   |   |   |   |

- **Control Characters**
- **Communication Control Characters**
- **Key Characters**
- **Second Category Representing Argument (Includes the Control Character DLE)**
- **Control Character DLE Included in the Argument Set**
The last argument offers the user the option of sending the message in 18 bit binary words (if ASCII is any non-zero constant) or as two (7-bit ASCII) 8 bit characters per word (right justified).

If the user requests the ASCII format, he must have a legal ASCII character in every character position of the send message buffer area indicated to the send routine and he must not use any of the 16 ASCII control characters in the field which includes "ETX". Specifically, no characters with format:

```
  b8           b1
   X 0 0 X 0 X X X
```

can be included within the text (in user's send buffer). The X's indicate "don't-care" bit positions. No problem arises in the send routine, but the receive routine must look for control characters and specifically the "ETX" to determine the end of the text. Refer to Figure 6 for ASCII code.

None of the above problem occur in the binary mode since the program sends only 6 bits of the 18 bit word at a time, and the ASCII bit positions "b8 b7" are forced to "0 1" at the send end and masked at the receive end when the message text is being processed. However, the program assumes that the user at the receive end knows whether the message is in ASCII or Binary format, i.e. this parameter is not passed to the receiver of the message. The user should utilize the "TO" and "FROM" parameter information to indicate which to the receiver.
As mentioned earlier (without explanation) the priority interrupt
facility should be disabled just before invoking the Send Message procedure.
This is because Send Message takes the 637 Interface out of "idle" mode
when it gets the go-ahead from the receiving end to send. At this point,
every character transmitted has to be a new character furnished by the
Send Message procedure on demand by the 637 Interface. Thus, the pro-
cedure has to be ready and waiting with the next character when the 637
has finished transmitting the current character. The procedure has one
character interval in real time (about 160 microseconds) to perform inter-
mediate fetching and formatting tasks to prepare the next character. This
is ample time unless an interrupt is permitted, then the timing is
indeterminate. If the response to the 637 Interface's request for the
next character is late, loss of line control follows. Initialization
will then be required before communications can be re-established.

Send Message Master/Slave Modification

Since it is possible for both parties to request to send at the same
time, some provision for Master/Slave priority must be written into the Send
Message routine. If the user is the Master, no modification is required. If
Slave, change the program in the SNDWAK routine by replacing the instruction:

    JMP SNDWAK

with two instructions:

    JMS RCVMES
    JMP SNDAGN
SUGGESTED PROGRAM MODIFICATIONS

In the course of preparing this description of the existing PDP-7/9 communications package, it was noted that certain channel conditions could possibly cause endless looping. Several minor program modifications, described below, should eliminate this danger, but have not been implemented. The suggested modifications are shown in the flow charts, and are marked with an asterisk to indicate a discrepancy between the charts and the actual program listing. One additional change is required if the functions of Master and Slave are reversed. These changes should be made in any future use of the package.

1. RECEIVE MESSAGE (1), near label RCVMES. Change TIMOUT error return from RCVDIE to a new subroutine called RCVTIM which alters the TIMOUT error return before proceeding to RCVDIE routine. This prevents an infinite loop condition in case the transmit routine always times out when called. RCVTIM is shown in RECEIVE MESSAGE (13).

2. RECEIVE MESSAGE (5), near label RCVOO. Add the instruction which increments NOSTX by 1 when "STX" is not found at the start of text.

3. RECEIVE MESSAGE (13), near label RCVDIE. Add the label variable REXIT, the subroutines RCVTIM and RCVEMG as shown to prevent loop condition described in change 1 above.

4. RECEIVE (3), near label RTXERR. Replace instruction JMS RCVDIE with JMS RCVEND.

5. SEND MESSAGE (2), near label SNDWAK. Add the modification per description on page 17 of report "Send Message Master/Slave Modification".
6. SEND MESSAGE (8), SNDTLM routine. Change TIMEOUT error return from
SNDTLM to a new subroutine called SENDMG and put modem into idle
mode before calling SNDEND. SENDMG prevents an infinite loop
condition in case the transmit routine always times out when
called. SENDMG is shown adjacent to SNDTLM and SEEXIT is a label
variable to be added as shown near subroutine SNDEND.
APPENDIX I

FLOW CHART
OF
COMMUNICATIONS
First the status parameters are zeroed.

PAREND contains (-) the number of locations between, but not including PARBEG and PAREND. These locations act as a named list of status parameters which are used and updated by this program.

PARBEG contains its own location. As used, it effectively points to the first status parameter location.

"X" is location 128 which has the property that it automatically increments its contents by one (underflowing) each time it is accessed indirectly, then the resulting indirect address is accessed.

All status parameters in the list are zeroed when the count in NOINT has reached 0.

Location NOINT is now the error return in case a timeout occurs during initialization. If a return is made here, the program will halt.

Initialize modem: See IOT Command Description for 637 Interface.

Setting up to send 600 = 35710 synchronizing characters (8 bits/character × 384 = 3072 bits) required by Bell 303 Modem.

SYN = 226, the 637 Interface synchronizing signal code for an 8-bit character position.
INITIALIZATION (2)

IOT to transmit SYN in order to get the transmit interface active (assuming to become active when SYN is first transmitted).

Now that interface is active, use normal routine to transmit SYN iteratively. See XCHAR routine.

Iterate through COUNT

IOT to repeat last character until the interface demands it.

Now look for his transmission.

Receive flag = 1? Y

Receive no more characters unless AC = SYN

Return to i(FINIT):

FINISH
**Action Routines Used By: **INITIALIZATION (3)

**NOINI: ENTRY**

+1 location of invoking instruction here

![Flowchart](image)

NOINI contains location of part of program which timed out if HALT occurs during initialization.

**RCVF: ENTRY**

+1 location of invoking instruction here

![Flowchart](image)

will wait until "SYN" is received before it returns.
**RECEIVE MESSAGE (1)**

Setup for providing arguments used by RECEIVE MESSAGE

1. Starting location of buffer where message is to be sent.
2. Length of buffer space available for message.
3. Location of user's receive schedule routine.

Return location pointer now points to user's normal return.

**RCVMES: ENTRY**

Receive Message routine to service interrupts or Send Message in slave mode.

* Suggest that RCVTIM be used as TIMOUT error return.

* RCVDIE is now TIMOUT error return.

Setup to read 8 characters

**RCVWAK:**

COUNT = COUNT + 1

COUNT = 0

Y 

RNOWAK:

NOENQ = NOENQ + 1

CALL RCVDIE

Error return

N

CALL RCHAR

Read the character

NORMAL RETURN

INTCH = AC

Store as last character received.

Make a note that no "ENQ" was found within 8 characters.
**RECEIVE MESSAGE (2)**

Success route: Setting up to RECEIVE MESSAGE.

Effectively moves buffer pointer back one location.

Now use auto index property of X as buffer pointer: \( X = RCVBUF - 1 \)

Put 13 complement of RCVCNT (length of buffer) into WRDCNT.

**ACK** = acknowledge, I'm ready to receive.

- **RCVACK**: Send "SYN" 3 times (previous page)
- **No Buffer space, Send "NAK"**
- **AC = ENQ ?**
- **AC = SYN ?**
- **AC = O ?**
- **RBDINT**: Bad Interface Leads to Error return
CALL XCHAR

Send it as character

AC = "NUL"

Now send "NUL"

CALL XCHAR

Repeat "NUL" when new character needed for transmission sync.

SET IDLE MODE

Try 8 character times to find start of header.

COUNT ← 8

Try his character

CALL RCCHAR

Read his character

AC = "30H"?

Y

Go to RCVGO

Success Route
Have found message start
of header. Will proceed.

N

COUNT ← COUNT + 1

N

COUNT = 0?

Y

NAMES:

No message header found.

NOMES ← NOMES + 1

CALL RCVDIE

Leads to error return.
**RCVGO:**

1. **SUM + AC**
2. **RCVERR = 0**
3. **CALL RBCHR**

- **GO TO RENDERR**

**Receive Message (4)**

- **Checksum begins with SOH. Now starting to process message.**
- **Clear RCVERR for this message.**

Read and sum next as binary character.

**Expected message number.**

Header error return. Leads to error return.

**Character appears to be message number. Now check the number against my record.**

**Subtract the message number in AC from RCVNO (my message number). RCVNO is zeroed by initialization and incremented each successful receive.**

- **AC < 0 ?**
  - **N**
    - **AC ← AC**
    - **CALL RDUROM**
    - **If MESNO < RCVNO, then must be duplicate of some old message.**
  - **Y**
    - **AC ← AC**

- **AC < 0 ?**
  - **N**
    - **CALL RDUROM**
    - **If MESNO = RCVNO, then this must be duplicate of last message.**
  - **Y**
    - **MESNO > RCVNO, then this is a new message.**

**Success Route**
RECEIVE MESSAGE (5)

(RCVGO CONTINUED)

CALL RECCHR

Expecting "TO" whom message is sent. Read and sum as binary character.

Expecting "FROM" whom message is sent as binary character.

Header error.

Header error.

Put in location "TO"

Put in location "FROM"

Add this instruction

Expecting start of text. Read and sum this as character.

Success Route
Begin receiving text.

* Add this instruction

GO TO RHDEFL

NOSTX ← NOSTX + 1

GO TO RHDEFL

Header error

AC = "STX" ?

N

Y

GO TO RCVTXT

ERROR

RETURN

NORMAL

RETURN

NORMAL

RETURN

ERROR

RETURN

GO TO RHDEFL

GO TO RHDEFL

GO TO RHDEFL

GO TO RHDEFL
**RECEIVE MESSAGE (G)**

**RDUPOM:**
- DUPOM → DUPOM + 1
- **Duplication of old message number indicated.**

**RDUPMS:**
- DUPMS → DUPMS + 1
- **Duplication of last message number indicated.**

**RCVERR → "ACK"**
- **RCVERR holds error reply; check sum is O.K. Will send acknowledge in this case.**
- Will process and checksum message since it is already on the way, but the text will not be stored since the message number indicates the message has been received previously.

**RCVTXT:**
- **Message header O.K. so far. Now determine how to process text. Read and sum next character.**

**CALL RSCHAR**
- **Text = ASCII; proceed**

**AC = "FS"?**
- **GO TO RCVASC**
- **Success Rules**
- **Text = Binary; proceed**

**AC = "6S"**
- **GO TO RCVBIN**

**RHDEERR:**
- **Bad Character in Header.**
- **BADCH ← BADCH + 1**
- **"DC2" = Bad Header reply;**
- **GO TO RCVOFF**
- **Leads to error return, but does needed housekeeping.**
RECEIVE MESSAGE (7)

Receive ASCII character. The Message Header has been processed successfully. Text will be processed as ASCII. Checksum maintained.

Expected Success Route when "ETX" is found.

Text return in 3-bit ASCII character (right justified) in AC.

Put this part in PARTCH: part of a character.

\[ x \leftarrow x + 1 \]
\[ z(x) \rightarrow AC \]

No more room in user's buffer space.

Get next character.

Possible Success Route, but part of character has not yet been stored in user's buffer.

"Exclusive Or" PARTCH with AC.
Effectively places first character (A) received adjacent to second (B).

Put ASCII word into user's buffer space after indexing buffer pointer.
RECEIVE MESSAGE (8)

Receive Binary character. The Message Header has been processed successfully. Text will be processed as Binary words. Checksum maintained.

Expected success route when "ETX" is found.

Text return has 6-bit binary character (right justified) in AC. Want to left justify first byte. Put this in PARTCH: part of a character.

Check word count when first byte of a new word is encountered.

Buffer overflow exit.

Get next binary character.

Possible success route. ETX found before last binary word completed.

Position second byte adjacent to first byte position before merging.

Merge this with current part of assembled binary word.
**RECEIVE MESSAGE (9)**

*(RCVBIN continued)*

- **CALL RBCCHR**

  - **ETX**
    - **RETURN**

  - **GO TO RCVLIST**

    - **TEXT**
      - **RETURN**

    - **AC ← PARTECH + AC**

    - **X ← X + 1**
      - **i(X) ← AC**

    - **GO TO RCVBIN**

*Possible success route: ETX found before last binary word completed.*

-Merge contents of PARTECH with last 6 bits of the binary word. Format of binary word below.*

- **Put the assembled binary word into user's buffer space after indexing buffer pointer.**

<table>
<thead>
<tr>
<th>5 16</th>
<th>11 12</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_0</td>
<td>B_1</td>
<td>B_2</td>
</tr>
<tr>
<td>B_3</td>
<td>B_4</td>
<td>B_5</td>
</tr>
<tr>
<td>B_6</td>
<td>B_7</td>
<td></td>
</tr>
</tbody>
</table>

---

**RCVOF: Buffer Overflow Routine.**

- **BUFFEROF ← BUFFEROF + 1**

- **AC ← "DC1"**

- **RCVERR ← AC**

- **CALL RCVENND**

*BUFFEROF contains number of Buffer Overflow events since initialization.*

-*"DC1" is reserved ASCII character 2218 = BUFFER OVERFLOW response.*

- **RCVERR contains the current response to receive error.**

*Leads to error return.*
**RECEIVE MESSAGE (10)**

Return address of invoking instruction here.

The message being sent is to be checksummed, but not formatted and placed in user's buffer space. Since this message may be ASCII (2CHAR/word) or Binary (3CHAR/word), allow 3 times as many characters to be flushed as WRDCUT allows for word space before indicating overflow.

Success Route for flushing this message. Now get checksum.

Buffer Overflow exit even though no buffer space was being used.

By convention, no more characters are flushed than worst case word count.
- 34 -

RECEIVE MESSAGE (11)

If entry is RCVLIST, put the portion of word already assembled into user's buffer area.

Expected entry is RCVSUM. Get 1st byte of checksum.

Expecting checksum as binary word. Skipping out all but rightmost 6 bits.

Assemble binary word for checksum. The checksum characters are not included in checksum.

Got 2nd byte of checksum.
RECEIVE MESSAGE (12)

(RCVSUM CONTINUED)

Checksum is now in AC.

Accept no more characters unless "SYN".

Reply "NAK" if checksum is not equal to SUM.

Error return

The checksum was OK, but there was some other error encountered. Error return.

No errors. Update message number.

At RCVACK: WRDCNT ← RCVCNT; the 15 complement of user specified buffer space. WRDCNT has been incremented after each assembled word. Want RCVCNT = message length.

Can only be minus zero (no message length)

Call user's scheduling routine with ARGS.

*Success Route* Send "ACK"
**RECEIVE MESSAGE (13)**

```plaintext
RCVEND: ENTRY
    CALL XCHAR
    CALL RCUDIE

NORMAL RETURN

RCUDIE: ENTRY
    AC ← "EOT"
    CALL XCHAR

*REXIT:
    SET IDLE MODE
    CLEAR RECEIVE ACTIVE
    GO TO i(RCVMES)

*RCVTUM: ENTRY
    TIMOUT ← R(RCVEMS)
    RTMOUT ← RTMOUT + 1
    GO TO RCUDIE

*RCVEMS: ENTRY
    GO TO REXIT
```

*See suggested program modifications*
RCHAR: ENTRY

TIME ← 408

RCH:

RECEIVE
ENDRCV=1?

N

RCH1:

TIME ← TIME + 1

N

TIME=0?

Y

READ A CHARACTER

RC ← AC

RETURN

N

RTMOUT ← RTMOUT + 1

CALL i(TIMOUT)

ERROR RETURN

RECEIVE (1)

Receive a character. Allow
~200 microseconds to respond

Line control signal has
terminated on modem. The
interface receive logic has
gone inactive.
Error return.

RECEIVE FLAG=1?

Y

The Interface has the
next character ready.

STORE this as "Last
Character Received"

SUCCESS route
Returns to i(RCHAR)

ERROR RETURN

No RECEIVE FLAG
indication within 200 msecs.
**RSCHAR:**

1. **ENTRY**
2. **CALL RSCHAR**
3. **NORMAL RETURN**
   - **MQ ← AC**
4. **SUM ← AC + SUM**
5. **AC ← MQ**
6. **RETURN**

**RECEIVE**(2)

- **Receive and sum a character.**
- **Get the character.**
- **Add this to sum.**
- **Replace character in AC**
- **Returns to i(RSCHAR)**

**RACHR:**

1. **ENTRY**
   - **Receive and sum ASCII character**
   - **Receive and sum the character.**
   - **Character is in MQ and AC.**
   - **Character in AC now: X XOX 000**
   - **Screen illegal ASCII characters in text.**
2. **CALL RSCHAR**
3. **NORMAL RETURN**
   - **AC ← AC "ISO"**
4. **AC = 0?**
   - **Y:** **RACHR:**
     - **AC ← MQ**
     - **AC = ETX?**
       - **Y:** **GO TO RXERR**
         - **ETX** Return: i(RACHR)
       - **N:** **RETURN**
8. **N:** **RETURN**
   - **Increment return pointer**
   - **Put orig. char. in AC**
   - **RETURN**
   - **Text Return: i(RACHR)**
Receive and sum Binary character
Receive and sum the character.

Store the character temporarily in RACHR.

Character in AC now: X0 XXX XXX
Screen illegal Binary characters in text.

Binary character received must be: 01 XXX XXX
Strip to: 00 XXX XXX

Increment return pointer

Text Return i (RBCHR)

Record text error.

Note type of error at termination.

Send "DC3" Bad character in message. Then go through termination protocol.

* This instruction replaces CALL RCVDIE.
- SEND MESSAGE (1) -

**SNOMES:**

**ENTRY**

**SNDTRY**

- 5

**SNDNO**

- (SNDNO + 1) MODULO 1000

**SNDAGN:**

**STO** = ARG1

**SFROM** = ARG2

**SBUF** = ARG3

**SWCT** = ARG2

**SBINT** = ARG5

**Send message arguments:** See Communications Package Protocol.

1. User number to whom message is sent.
2. User number from whom message is sent.
3. Starting location of buffer where message exists.
4. Length of buffer as word count of message.
5. O = ASCII, else message is binary.

Return location pointer now points to calllist error return.

**TIMOUT** = 2(SNDTIM)

**CALL XCLSYN**

**NORMAL RETURN (XCLSYN)**

**AC** = "ENQ"

**CALL XCHAR**

**Normal Return (XCHAR)**

**SET IDLE MODE**

**COUNT** = -3000

**SNOWK:**

**RECEIVE**

**FLAG** = 1 ?

**COUNT = COUNT + 1**

**COUNT = 0 ?**

**Y**

**CALL SNERR**

(Never returns)

**Success Route**

GO TO SNOWK

(Next page)

**Set up to allow him 10 more to answer**

**No answer**

**NORM**

**NORM**

- NOANS + 1

**CALL SNERR**

(Never returns)
Send Message (2)

SNDWAK:

CALL RCHAR

NORMAL RETURN (RCHAN)

WCHR ← AC

AC = "ACK"

Y

GO TO SNDLOG

N

AC = "WAK"

Y

NOBUFS ← NOBUFS + 1

N

CALL SNDERR

SNOWAK:

CALL RCHAR

Get his answer from read character routine into AC, then see what it is.

Put it in "wake character" for the record.

Success Route:

He has acknowledged to receive my transmission. Time to send it.

He has no buffer area available.

Error return.

No sends first?

Y

Am I master?

Y

CALL RCVMES

N

COUNT ← COUNT + 1

GO TO SNDAGAIN

COUNT = 0?

N

NOBUFS ← NOBUFS + 1

CALL SNDERR

Him

N

CALL RCVMES

N

COUNT = 0?

Y

Couldn't wake him up in time.

Neve returns, but return location of calling instruction is stored here for reference.

* The program must be modified to establish the SLENVE status.

He wants to send a message, too?

His message none of these. Get next character if time permits.
SEND MESSAGE (3)

Now attempting to send message. Begin by clearing checksum.

Put on message header.
SOH = 20h : Start of header sent first

Transmit and sum character in AC.

Want to send a new character every interval now. Would rather muddle loose sync than duplicate previous character, if I can't supply the next one on time.

Message number follows "SOH".
Transmit word in AC as 18-bit binary number (send as 3 8-bit characters - see XBCHR) and old to checksum.
User number who is to receive it.

Send as binary.
User number who is sending it.

Send as binary.

STX = 202h : Start of text

Send as character.
Set up for sending text.
Put buffer pointer (less 1) into auto-indexing location X.

1's complement of SWDCT (length of buffer) into WRDCNT.
- SEND MESSAGE (4) -

(SNDGO CONTINUED)

If SNINF = 0?

N = Binary text

Y = ASCII text

AC ← "FS"

FS = 234g: Sending ASCII

CALL XSCAR

Send as character

AC ← GS

GS = 235g: Sending binary

CALL XSCAR

Send as character

X points to Sender's previous word.

X ← X + 1

AC ← e(X)

CALL XAWRD

Format and send as ASCII word.

WRDCNT ← WRDCNT + 1

N ← 0?

X ← X + 1

AC ← e(X)

CALL XBWRD

Format and send as binary word.

WRDCNT ← WRDCNT + 1

N ← 0?

SNDETX:

AC ← "ETX"

ETX = 203g: Send end of text

CALL XSCAR

Send as character

Get the checksum and send as binary word.
SEND MESSAGE (5)

3

CALL XBWRD

Format and send as Binary Word

AC = "NUL"

NUK = "200" Send null character until he responds.

CALL XCHAR

Send one character

N.R.

SET IDLE MODE

Repeat last character when needed.

COUNT = -7

Set up to wait 7 character intervals ≈ 112 msec.

SNDWT:

CALL RCHAR

Read his response character

N.R.

RSPCH = AC

Put into RSPCH for record. Then see what the response is.

AC = "ACK"?

Success Route
He acknowledges a good message.

Y

GO TO SNDWR

N

AC = "NUL"?

Bad checksum in message. Will try again. SNDWR +3.
SEND MESSAGE(c)

(SNOWT CONTINUED)

- 45 -

AC = "D1" ?
Y: GO TO SBUF0F
N: Buffer overflow. Will not try again.

AC = "D2" ?
Y: GO TO SBADCH
N: Bad character in header. Since checksum was OK, user must have made the error. Will not try again.

AC = "D3" ?
Y: GO TO SBADCM

COUNT = COUNT + 1

COUNT = 0 ?
Y: SNORSP
N: Go to SNOWT

SNORSP: NORSP ← NORSP + 1

CALL SENDRR

No response. Make a note of it and try again, unless SNDTRY = 0.
Action Routines Used By:

**SEND MESSAGE**

**SBDSM:**

```
BAD3MS+1
```

- **Bad checksum**

CALL **SNDErr**

- Never returns, but location of instruction that invoked it is stored here for reference.

**SNDErr:** Entry

- Location of invoking instruction here.

```
SNDFTRY+1
```

- Branch here when send error is found which allows SNDFAGN if SNDFTRY ≠ 0.

```
SNDFTRY = 0?
```

- N: GO TO SNDFAGN

- Y: Tried 5 times and lost.

CALL **SNDFEND**

**SOFOF:**

```
BUFOFS+1
```

- Buffer overflow

CALL **SNDFEND**

- Error return

**SBACH:**

```
BADCHS+1
```

- Bad character in header

CALL **SNDFEND**

- Error return
Action Routines Used By: SEND MESSAGE (8)

**SBADCM:**
- BADCMS → BADCMS +1
  - Bad character in message.
- CALL SUDEND
  - Error return

**SNDTMS:** ENTRY
- Location of invoking instruction here.

**TIMOUT→ 8(SNDEMG)**
- Change TIMOUT return to emergency exit.

**STMOUT→ STMOUT +1**
- Time Ran Out in transmission attempt.

**SET IDLE NODE**
- CALL SUDEND
  - Error return

**SNOOK:**
- SNDMES → SNDMES +1
  - Success route
  - Return location is at SNDMES. Pointer has been at users error return location. Since success route has been found in send message routine, increment this location by one so as to point at users normal return location.
- CALL SUDEND
  - Normal return

**SNDEND:** ENTRY
- Location of invoking instruction here.

**AC→”EOT”**
- Send end of transmission

**SEXIT:**
- Normal Return (KCHAR)
- Clear Receive Active
  - Accept no more characters unless = SYN.
- RETURN
  - Return to 8 (SNDEMS), either points to users error return or normal return.
**TRANSMIT (1)**

+1 location of invoking instruction here.

Transmit the rightmost 8 bits of the AC
Record as last character sent.

Set up to allow 200 ms seconds
  to communicate with interface.

TXCHAR: ENTRY

\[
\text{TIME} \leftarrow 400
\]

\[
\text{AC} \leftarrow \text{AC} + 3778
\]

\[
\text{SUM} \leftarrow \text{AC} + \text{SUM}
\]

\[
\text{RETURN}
\]

\[
\text{HALT}
\]

\[
\text{RETURN}
\]

TXCH:

\[
\text{IF TRANSMIT FLAG = 1}
\]

\[
\text{AC} \leftarrow \text{AC} + \text{SUM}
\]

\[
\text{RETURN}
\]

TXCH:

\[
\text{IF TIME = 0}
\]

\[
\text{CALL i(TIMOUT)}
\]

Failure Route

Run out of time.

Send the character plus clearing
  the transmit flag will cause next instruction
to be skipped if successful.

If return from interface is here, the program halts.
  Can only happen if interface goes inactive (loses sync)

Success Route

Returns to i (TXCHAR). Character is being sent
  and is still in AC.

Return is to current TIMEOUT routine.
TRANSMIT (2)

XAWRD: ENTRY

+1 location of invoking instruction here.

Transmit an ASCII word. Assume 2 8-bit right justified characters in AC. Send leftmost one first. Format below.

AC(0)

MQ(0) = DONT CARE

AC(0)

MQ(0) = DONT CARE

AC(0)

MQ(0) = DONT CARE

AC(0)

MQ(0) = DONT CARE

AC(0)

MQ(0) = DONT CARE

AC(0)

MQ(0) = DONT CARE

Send and sum character.

Returns to e(XAWRD)
- 50 -

**TRANSMIT(3)**

1. Location of invoking instruction here

Transmit a Binary word. Send as 3 8-bit characters using 6 bits of AC in each character. Send leftmost 6 bits of AC first.

Format below:

```
AC(0) | B_0 B_1 ... B_5 B_6 B_7 |
MQ(0) = DON'T CARE
```

```
AC(1) | X ... DON'T CARE ... X X B_6 B_7 B_8 B_9 B_10 B_11 |
MQ(1) | B_2 B_3 ... B_6 X ... X |
```

Sending B_0 - B_5

```
AC(2) | X ... DON'T CARE ... X X B_6 B_7 B_8 B_9 B_10 B_11 |
```

Sending B_6 - B_11

```
AC(3) | X ... DON'T CARE ... X X B_6 B_7 B_8 B_9 B_10 B_11 |
```

Sending B_12 - B_17

Normal return

Returns to i(XBWRO).
**TRANSMIT(4)**

**XBCHR:**

- **ENTRY**
- \[ AC \leftarrow AC \& \text{"77"} \]
- \[ AC \leftarrow AC \oplus \text{"100"} \]
- **CALL XSCAR**

**RETURN**

- +1 location of invoking instruction here

**Transmit a Binary character**

- Strip off all but rightmost 6 (least significant) bits of AC by "ANDing" AC with "77\text{"}"

- Map this into ASCII non-control type characters by "EXCLUSIVE ORing" AC with "100\text{"}"

- Transmit and sum as character.

**Binary Character**

**XCLSYN:**

- **ENTRY**
- \[ AC \leftarrow \text{"SYN"} \]
- **CALL XCHAR**

**NORMAL RETURN**

**CALL XCHAR**

**NR.**

**RETURN**

- +1 location of invoking instruction here

**Clear line and sync**

- \[ SYN = 226_{8} : \text{Convention to send "SYN" 3 times before message transmission.} \]

- Send as character.

- "SYN" is still in AC.

**Returns to \( i'(XCLSYN) \)**
COMMUNICATIONS PACKAGE ERROR PARAMETERS

Receive Errors in Message

Each time an error occurs within the Receive Message Procedure, the source of error is indicated by adding "1" to one of the following parameters.

BADCH  Bad character in header; produced in header processing, if an "ETX" was found during, or "STX" was not found after, the header was processed. Note that other types of header errors are possible.

BADCM  Bad character in message; not used (see TXERR).

BADINT Bad character after "SYN"; produced when a character other than "ENQ" followed the "SYN" sequence at beginning of received message.

BADSUM Bad checksum; produced when checksum maintained by Receive Message Procedure did not match checksum received after "ETX" of message.

BUFOVF Buffer overflow; produced when message text received exceeded size of buffer space allocated. The part of the message that did not fit was lost.

DUPMS  Duplication of last message; produced when message number contained in header was same as last message number. Message numbering is maintained by the Communications Package.

DUPOMS Duplication of old message; produced when message number contained in header was smaller than last message number.

INTCH  The last character received before an error return due to NOENQ or BADINT. Otherwise, INTCH contains the "ENQ" character.

NOBUF  No receive buffer; produced when user did not provide RCVSET with a buffer, after response to "ENQ", request to send a message.

NOENQ  No "ENQ"; produced when "ENQ" does not follow "SYN" within 8 characters of the "SYN" sequence at beginning of received message.
No message header found; produced when no "SOH" was received within
8 characters after sender's "ENQ" was received and acknowledged at
beginning of received message.

No "STX" received after message header was processed.

Text error; produced while receiving text if a control character
other than "ETX" was received.

Send Errors in Message

Each time an error occurs within the Send Message Procedure, the source
of error is indicated by adding "1" to one of the following parameters.

Bad character in message; produced if received response to message
after it was sent was "DC3" indicating receiver found bad character
in message.

Bad character in header; produced if received response to message
after it was sent was "DC2" indicating receiver found bad character
in header.

Bad sum; produced if received response to message after it was sent
was "NAK" indicating receiver's checksum did not match SUM sent as
binary word.

Buffer overflow; produced if received response to message after
it was sent was "DC1" indicating receiver's buffer overflowed.

No answer; produced before message was sent if no "SYN" sequence was
received within 10 milliseconds of sending an "ENQ," request to send
message.

No buffer at receive end; produced before message was sent if "NAK"
was received in response to "ENQ," request to send message.

No response; produced after message was sent if no legal response
character was received within 7 character intervals after "NUL"
send sequence.

NOWAK
No acknowledge after "SYN"; produced before message was sent if received response was "SYN" sequence, but not followed by legal "ACK," "NAK" or "ENQ" replies within 10 millisecond response time allowed.

RSPCH
The response character received after a message was sent. If the message was sent successfully this character will be "ACK," otherwise, the type of error indicated should identify RSPCH.

STMOUT
Timeout occurred while in Send Message Procedure. The timeout may have been due to attempt to transmit or receive. See Error Parameters Common to Send or Receive Message Procedures.

WAKCH
The wake character received before a message was sent. It is the character received in response to "ENQ," request to send message unless NOANS was indicated.

Error Parameters Common to Send or Receive Message Procedures

ENDRCV
Receive End Flag came on; produced when the Modem has lost line control (usually due to lost synchronization) in RCHAR routine attempting to read next character.

RTMOUT
No Receive Flag indication before timeout occurred; produced when Interface did not signal that it has received a character within 200 microseconds after entry into RCHAR routine (usually due to Interface in Receive Inactive state and no "SYN" sequence arrives to activate it).

RC
The last character successfully received by the RCHAR routine.

WC
The last character sent to XCHAR routine to be transmitted.

XTMOUT
No Transmit Flag indication before timeout occurred; produced when Interface did not signal that it was ready to transmit a character within 200 microseconds after entry into XCHAR routine (usually due to loss of sync).