Burroughs

Reference Manual

B 6000/B 7000 Systems

NDL

(Relative to Release Level 3.5)

Distribution Code SE

Priced Item
Printed in U.S.A.
June 1985

1176690

Burroughs

Reference Manual

B 6000/B 7000 Systems NDL

(Relative to Release Level 3.5)

Distribution Code SE

Priced Item
Printed in U.S.A.
June 1985

1176690

Burroughs cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued from time to time to advise of such changes and/or additions.

Correspondence regarding this publication should be forwarded, using the Documentation Evaluation Form at the back of the manual, or remarks may be addressed directly to Burroughs Corporation, Corporate Product Information East, 209 W. Lancaster Ave., Paoli, PA 19301, U.S.A.

Burroughs

PUBLICATION CHANGE NOTICE

PCN No.: 1176690-001 Date: December 30, 1985

Publication Title: B 6000/B 7000 Systems NDL Reference Manual (6/85)

Other Affected Publications: none

Supersedes: n/a

Description

The pages provided with this PCN contain changes to the B 6000/B 7000 Systems NDL Reference Manual, form 1176690, dated June, 1985. The locations of changes are indicated by black vertical bars on the replacement pages.

Replace These Pages

iii
5-141

LIST OF EFFECTIVE PAGES

Page	Issue
iii, iv	PCN-001
v thru xii	Original
1-1 thru 1-9	Original
1-10	Blank
2-1 thru 2-3	Original
2-4	Blank
3-1 thru 3-17	Original
3-18	Blank
4-1, 4-2	Original
5-1 thru 5-140	Original
5-141, 5-142	PCN-001
5-143 thru 5-206	Original
6-1 thru 6-21	Original
6-22	Blank
A-1 thru A-4	Original
B-1 thru B-5	Original
B-6	Blank
C-1, C-2	Original
D-1 thru D-11	Original
D-12	Blank
E-1 thru E-5	Original
E-6	Blank
F-1 thru F-3	Original
F-4	Blank
1 thru 12	Original

LIST OF EFFECTIVE PAGES

Page	Issue
iii	Original
iv	Blank
v thru xii	Original
1-1 thru 1-9	Original
1-10	Blank
2-1 thru 2-3	Original
2-4	Blank
3-1 thru 3-17	Original
3-18	Blank
4-1, 4-2	Original
5-1 thru 5-206	Original
6-1 thru 6-21	Original
6-22	Blank
A-1 thru A-4	Original
B-1 thru B-5	Original
B-6	Blank
C-1, C-2	Original
D-1 thru D-11	Original
D-12	Blank
E-1 thru E-5	Original
E-6	Blank
F-1 thru F-3	Original
F-4	Blank
1 thru 12	Original

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

LIST OF ILLUSTRATIONS

LIST OF TABLES

PURPOSE

The Network Definition Language (NDL) is a high-level programming and definition language that is used to describe a data communications network physically, logically, and functionally. It has been implemented for use on B 6000 systems that use Data Communication Processors (DCPs) within the Multiplexer (MPX) I/O subsystem or B 7000 systems that use DCPs within the I/O Modules (IOMs).

This manual is intended as a reference manual for the Burroughs NDL. Its purpose is to present a complete description of the syntax and semantics of all language components and compiler options of NDL within a framework that is designed for quick access of information. An overview and functional description of the DCP-based datacomm subsystem in which NDL programs run, is also presented.

The reader is assumed to be an experienced datacomm programmer with a detailed knowledge of Burroughs B 6000/B 7000 Series datacomm and of general datacomm concepts. These should include, but not be limited to, terminal addresses, line speeds, vertical and horizontal parity, line polling/select procedures, and terminal hardware characteristics. A knowledge of Message Control System (MCS) concepts and of the DCALGOL language would also be helpful.

The notation used in this manual to represent the NDL syntax is the railroad syntax diagram, commonly used in Burroughs manuals. For those unfamiliar with this notation, a complete description is provided in Chapter 2, NDL Syntax Conventions.

INTRODUCTION

The Network Definition Language (NDL) is a structured, user-oriented datacomm implementation language for the B 6000 and B 7000 systems.

NDL is used for B 6000 systems utilizing Data Communications Processors (DCPs) within the Multiplexer (MPX) I/O subsystem. NDL is also used for B 7000 systems that use DCPs within the I/O Modules (IOMs). The use of NDL on the B 6000/B 7000 systems is equivalent to the Network Definition Language (NDLII) for the A Series and B 5900/B 6900/B 7900 Series systems that use Network Support Processors (NSPs), Line Support Processors (LSPs), and Datacommunications Data Link Processors (DCDLPs) within the I/O Data Communications (IODC) subsystem.

The NDL source code supplies the NDL compiler with information to produce the proper Network Information File (NIF) and DCP Code File (DCPCODE) required to operate the defined network.

NDL provides the high-level language constructs required to fully utilize the hardware capabilities of the DCP subsystems, including bit-oriented protocols, full-duplex communications and exchange and reconfiguration features.

NDL can implement sophisticated datacomm networks without requiring cumbersome programming or detailed knowledge of machine hardware.

SOURCENDL is an NDL program that provides an example of how an NDL program can be written.

This document provides reference data for the experienced programmer who is familiar with the B 6000 and B 7000 Data Communications System.

This reference manual is divided into the following six chapters and six appendices.

- Chapter 1, **INTRODUCTION**, describes where the **NDL** Manual fits into the existing Data Communications System documentation, and defines the scope of **NDL.**
- Chapter 2, **NDL SYNTAX CONVENTIONS**, explains the syntactical notation used in defining the Network Definition Language.
- Chapter 3, **LANGUAGE COMPONENTS**, describes the elements that form the most primitive structures of the language.
- Chapter 4, **SOURCE PROGRAM STRUCTURE**, describes the basic structure of an **NDL** program.
- Chapter 5, **DEFINITIONS**, describes the various definitions that make up an **NDL** program.
- Chapter 6, **VARIABLES**, describes the program variables available to the **NDL** programmer.
- Appendix A, **RESERVED WORDS**, is a list of "words" that have been set aside for specific purposes within the Network Definition Language.
- Appendix B, **TRANSMISSION CODES**, provides useful data transmission code tables.
- Appendix C, **SOURCE INPUT FORMAT AND CODING FORM**, describes the input format and coding form to be used by the programmer.
- Appendix D, **COMPILE-TIME OPTIONS**, describes the compiler options available to the user.
- Appendix E, **COMPILER SOURCE AND OBJECT FILES**, describes how compiler communication is handled through various input and output files.
- Appendix F, **FULL DUPLEX**, describes full duplex and the NDL statements associated.

For additional information, refer to the following documents:

Title	Form No.
B 6700/B 7700 Data Communications (DATACOM) Functional Description	5000060
Input/Output Subsystem Reference Manual	5014483
B 6000/B 7000 Series Message Oriented Data Communications (MODC) Information Manual	5011836
DCALGOL Reference Manual	5014574

NOTE

For Adapter Cluster Model III (ACIII) information, refer to the (MODC) Information Manual (5011836).

CHAPTER 1

NDL DESCRIPTION

SCOPE OF NDL

The Network Definition Language (NDL) source program describes a data communications network physically, logically, and functionally. Physical components of a data communications network include the hardware specifications and capabilities of the various elements which comprise the network. The logical characteristics of a network are the associations among the various components of a data communications subsystem (user programs, Message Control Systems, etc.), application-oriented characteristics (page size and width, special-purpose characters, etc.), and the symbolic names used to reference physical elements within the network. Table 1-1 lists the physical and logical characteristics in their relation to the network elements. Figure 1-1 illustrates the spere of influence of a NDL source program on the logical and physical components of a Data Communications System. The area enclosed by the broken line is the "global sphere of influence." The shaded area indicates the "local sphere of influence." The arrows indicate the flow of information.

NDL also specifies the functional behavior of the network or the way in which each data communications line is to be controlled. These specifications consist of individual routines, allowing the NDL programmer to implement the protocol required to meet the physical characteristics and applications of the types of terminals that have been defined. The routines are compiled into a set of instructions which the Data Communications Processor (DCP) executes to perform the functions described by the NDL program.

The NDL source program is transformed into two files containing the information required to operate the defined network:

a. The Network Information File (NIF), containing the logical and physical specifications of the network.

b. The DCP Code File (DCPCODE), containing the Data Communication Processor (DCP) hardware instructions for operating the network.

USE OF NDL

Once the data communications hardware has been installed on a B 6000/B 7000 system, several software systems are required to generate and operate the data communications network. These packages, illustrated in figure 1-1, consist of one or more Message Control Systems (MCS), the Data Comm Controller (DCC), and the NDL compiler. The purpose, function, and use of each of these software items are described in the following paragraphs.

Table 1-1. Network Characteristics

Network Elements	Physical Characteristics	Logical Characteristics
DCPs	Memory size Reconfiguration capabilities	Set of terminals controlled by each DCP
LINEs	Physical location (address) Transmission speed Type of line and connection	Station line assignments Automatic answer capability
MODEMs	Physical delays Transmission speed and type Continuous vs. controlled carrier	Symbolic name
STATIONs	Terminal device characteristics	Symbolic name Logical attributes Associated Message Control System
TERMINALs	Transmission code, speed and type Parity	Transmission numbers Special characters

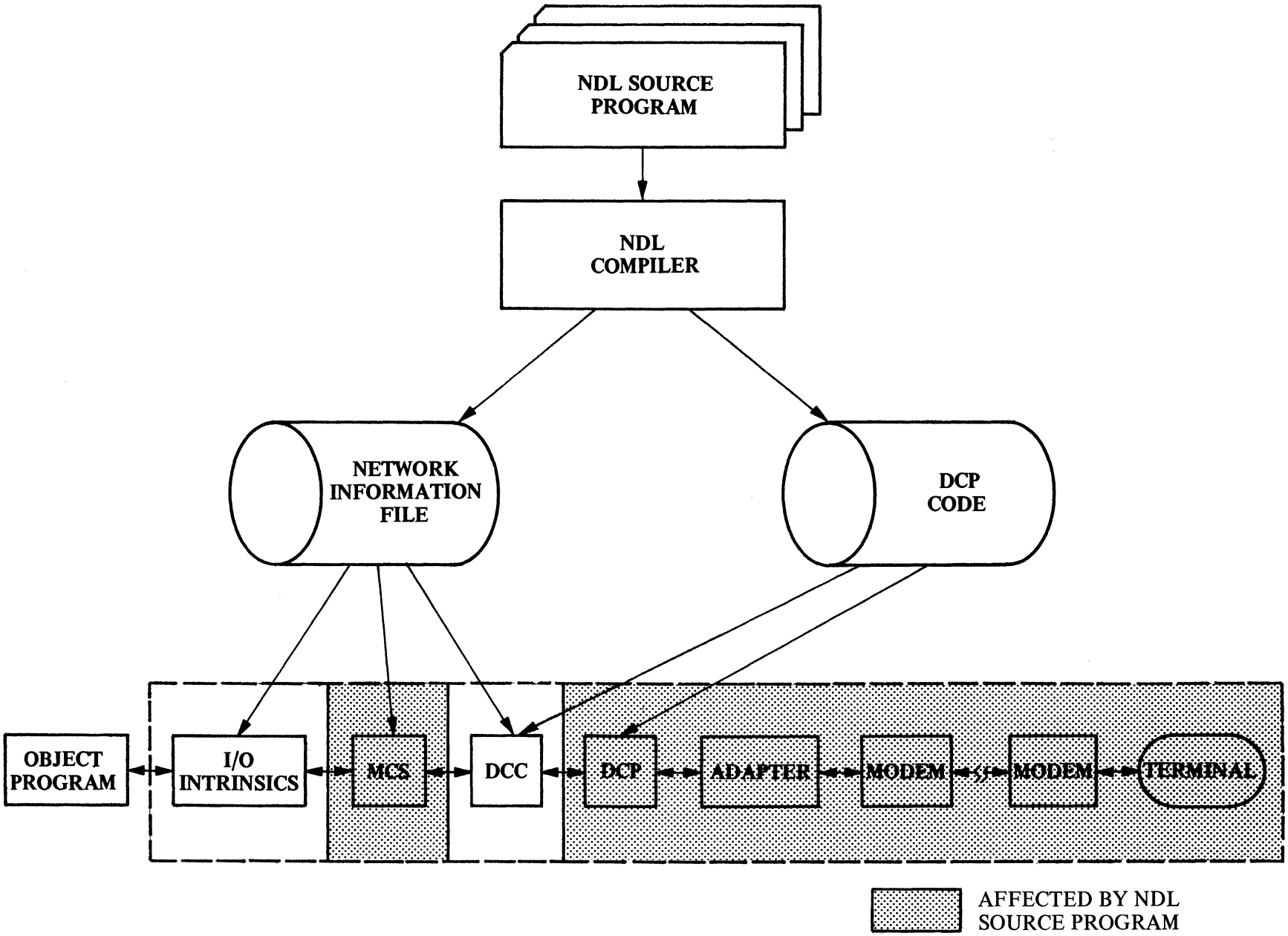


Figure 1-1. NDL Sphere of Influence

Message Control System (MCS)

An MCS is a special purpose DCALGOL program which may be a Burroughs-supplied MCS such as SYSTEM/CANDE, SYSTEM/RJE, SYSTEM/DIAGNOSTICMCS, SYSTEM/COMS, or a user-written program. The primary function of an MCS is, as its name implies, to control the flow of data communications messages between the terminal and the main system. Information from the DCP, such as terminal input and status information, is forwarded to the MCS via the DCC. Messages from the MCS to the DCP, such as terminal output or network changes, are performed by the MCS invoking an intrinsic function called DCWRITE. Each station which is defined by the NDL source program must have one, and only one, controlling MCS.

Data Comm Controller (DCC)

The DCC is the basic interface between the DCP and the main system. It exists as a subset of the basic B 6700/B 7700 Master Control Program (MCP) and operates as an independent task or stack, one such task for each active DCP.

Before a defined data communications network may be utilized, the DCPs which comprise the network must be initialized. As each DCP is initialized, the portion of the NDL-defined network which utilizes that DCP becomes active.

Once initialized, each DCC stack transfers messages between the associated DCP and the proper MCS.

NDL Compiler

Whereas the DCC and an MCS are required to operate a data communications network, the NDL compiler is used to generate the tables and DCP code which, to a large extent, control the way in which the network functions.

The NDL source program, then, must supply the NDL compiler with information which will allow the compiler to produce the proper NIF and DCPCODE files to operate all of the Data Comm Processors and their sub-components within the network. (In figure 1-2, the shaded areas indicate the areas of the data communications subsystem which are influenced by the NDL source program.) Although an NDL program may contain up to 11 discrete sections, it functionally consists of two interdependent pieces of information: the network description and the DCP programs.

NETWORK DESCRIPTION

The NDL programmer uses various sections of the NDL source program to describe the logical and physical characteristics of the network. The information supplied in those sections is used, in part, to supply the DCC with the proper tables and DCP code that are used to operate the network.

The NDL compiler performs consistency checks across the various definitions to ensure that the defined network is logically structured. For example, a line must not be associated with a particular modem if the defined speed range and transmission type of the modem do not permit a proper interface to the line. Similarly, a terminal defined to operate in an asynchronous mode must not be associated with a line which uses a synchronous adapter.

All of the information supplied by the NDL definitions is recorded within the NIF file. This enables an MCS or user program to gain access to many of the logical characteristics of the network, as well as permitting dynamic reconfiguration of the network by an MCS.

These definitions are also used to modify or include special areas of DCP code which are network dependent. For example, the DCP code for transmitting or receiving characters on a synchronous line is different from such code for transmitting or receiving on an asynchronous line. In addition, if any dial-out type lines are defined within the network, extra code must be generated for performing dial-out functions. Thus, the NDL compiler "tailors" the resultant DCPCODE file to fit all the requirements of the defined network.

DCP PROGRAMS

Once a data communications network is logically and physically defined, the functional operation of each line and station within the network must be described. These descriptions, called **CONTROL** definitions and **REQUEST** definitions, are individual programs which are executed by the DCP when required to perform the necessary line discipline. Each line must have one associated **CONTROL** definition, and each terminal may have one or more associated **REQUEST** definitions.

The **CONTROL** and **REQUEST** definitions consist of NDL statements which the compiler transforms into the DCP instructions to be executed when performing a particular network function. A **RECEIVE REQUEST** definition is invoked when input from a terminal is to be processed, and a **TRANSMIT REQUEST** definition is executed when output to a terminal has been requested by an MCS or user program. The line **CONTROL** definition is utilized to determine when and for which of the stations on the line a **REQUEST** definition is to be executed.

Since an NDL source program must handle many lines, the DCP must share its processing capabilities among the lines it services. Due to the fact that the data communications subsystem operates in a real-time environment, few network functions, if any, require the dedicated use of the DCP for an extended length of time. A **RECEIVE REQUEST**, for example, usually spends most of the time waiting for a character to be sent from a terminal. Likewise, a **TRANSMIT REQUEST** can only operate as fast as the line speed permits. Thus, while a **REQUEST** definition is waiting for an external event, or interrupt, from a line, the DCP is free to continue execution of a **REQUEST** or **CONTROL** definition for another line.

The allocation of the DCP for the servicing of its many lines is one of the duties of the basic DCP operating system and the **CONTROL** definitions. Figure 1-2 illustrates the means by which the control of the DCP is transferred between the operating system and the **CONTROL** and **REQUEST** definitions.

Line Control

Each **CONTROL** definition, or "Line Control Procedure," must perform two functions. First, it must select which station on the line is to receive attention next, and second, it must decide what particular function is to be performed for that station. If the function to be performed is an output request, control is transferred to the **TRANSMIT REQUEST** for the station. If the function is an input operation, the station's **RECEIVE REQUEST** is executed. Network functions which do not involve the reception or transmission of messages, such as status or network changes, are performed by invoking a common subprogram, or macro, within the DCP operating system itself.

Request Definitions

For each type of terminal which is capable of output, a **TRANSMIT REQUEST** must be named within the terminal definition. Likewise, if a terminal has input capabilities, a **RECEIVE REQUEST** must be supplied and named. Typically, many stations may share the same **REQUEST** definitions, just as many lines may utilize the same **CONTROL** definition. In some cases, more than one set of **REQUEST** definitions may be desired, and defined, for a station.

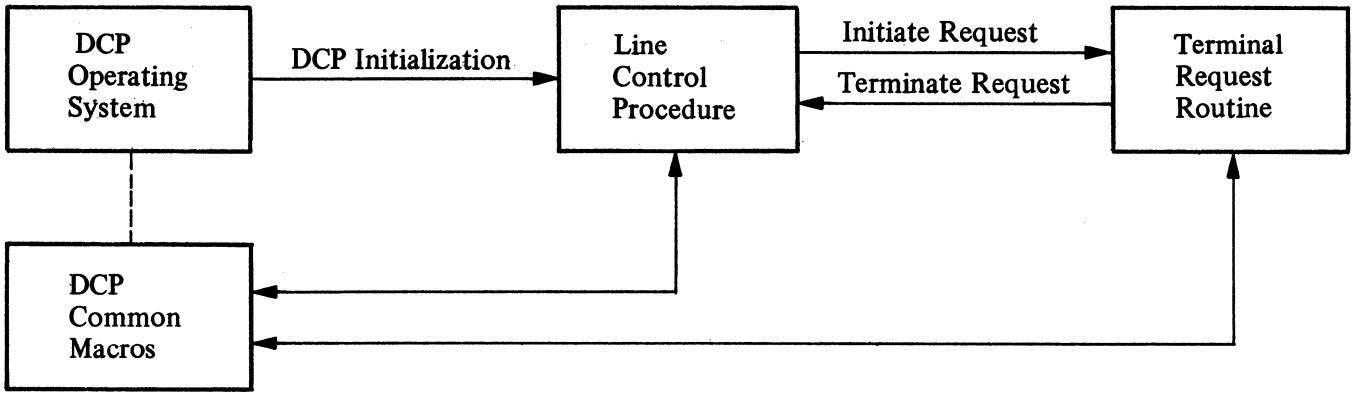


Figure 1-2. Transfer of Control Within the DCP

The functions of a **REQUEST** definition may be as simple or as complex as the application of the station dictates. One of the basic design goals of the DCP is to free the main system from the burden of performing basic terminal receptions and transmissions. However, by the proper application and coding of the **CONTROL** and **REQUEST** statements in NDL, a significant amount of intelligent message processing may be performed by the DCP, thereby allowing more of the main system's resources to be free to perform other work.

The NDL programmer must keep in mind, however, that the DCP runs at a finite rate, and that it is operating in real-time. Thus, if too much time is spent processing a message, other lines may fail to be serviced quickly enough to avoid transmission errors. Several NDL statements are provided to "break up" long strings of NDL code to ensure that the DCP may properly service all of its lines.

When a **REQUEST** definition has terminated the processing of an input or output function, it usually branches back to the beginning of the **CONTROL** definition. The **CONTROL** definition then selects the next station to be serviced and the process continues.

Thus, the functioning DCP can be visualized as a small multiprogramming system, where each line has its own program and operating environment and runs asynchronously and independently of the other lines. The **CONTROL** definition and its associated **REQUEST** definitions form the "main program" for each line, and the common DCP macros are "sub-programs."

DCP INTERACTION WITH THE MAIN SYSTEM

Although the DCP is a self-contained and asynchronous device with respect to the main system, it is not an autonomous unit, and requires the active participation of the main system and its resources to properly function. In particular, main memory storage space is required to contain tables and messages. In addition, the DCP requires the allocation of a pool of message areas in main memory for the gathering of input from terminals and reporting of error conditions.

DCP Tables

The NDL compiler constructs a series of tables which reflect the physical and logical characteristics of the network as defined by the NDL source program. The DCP uses these tables for the storage of status information, and for determining what types of functions are to be performed for each of the many different lines and stations which the DCP controls.

The compiler places a disk image of these tables within the DCPCODE file along with the DCP code itself. When the DCP is initialized, the tables are loaded into main memory by the DCC, which also provides the

DCP with a reference to the tables. If several DCPs exist which share hardware-exchanged adapter clusters, two DCPs may utilize the same set of tables if the network description indicates this mode of operation. In the case where a DCP is not "exchanged" in this manner, each DCP uses its own unique set of tables.

Each set of tables can be divided into two sets of information. Each line has a table, and each station has a table. The DCP uses a "line descriptor" to reference each line table. The descriptors for all lines controlled by the DCP are stored within a vector, which is then indexed by the physical line adapter address. Each line descriptor contains information concerning the status of the line (not ready, connected, busy, etc.), physical characteristics of the line (dial-out, switched, etc.), logical characteristics (automatic answer, etc.), and a reference to the **CONTROL** definition which is used for the line. In addition, the line descriptor contains the memory address of the actual line table.

Each line table contains additional information describing the logical and physical characteristics of the line. Much of the information in the line table can be referenced and/or modified directly by the NDL **CONTROL** and **REQUEST** definitions. Other information is reserved for use by the DCP operating system.

Immediately following the line information in the line table is a vector of station descriptors, one such descriptor for each station which can exist on the line. Similar to the line descriptor, each station descriptor addresses a table of information for a particular station. The DCP references the proper station table by indexing into the line table by the proper relative station address, or "station index," and using the addressed station descriptor to reference the proper station table. When a line **CONTROL** Procedure "selects" a station for the purposes of initiating a **REQUEST** definition, it is actually selecting the proper station table for use. Just as the line table contains a reference to the proper **CONTROL** definition to execute, the station table contains references to the appropriate **TRANSMIT** and **RECEIVE REQUEST** definitions. It should now be apparent that each station assigned to a particular line must utilize the same Line Control Procedure, since the stations on the line all share a common line descriptor and line table. However, each station may have a different set of **REQUEST** definitions, since these routines are station oriented.

Each station table is of a fixed size; however, this size is determined at NDL compilation time. The fixed size is between six to ten words, depending on the constructs which are used. For example, when DCP sequencing is used, the station table size is eight words. Certain message-oriented constructs for autonomous DCP systems (such as editing and filemode) require additional words in the station table.

Although each station table is of a fixed size, the line tables will vary in size directly proportional to the number of stations which can potentially exist on the line. The NDL source program specifically defines the maximum number of stations for each line, as well as which stations are assigned to what line. Not all station descriptors may be utilized for a given line, i.e., the number of real stations on a line at any given moment may be less than the true capacity of the line. A line may be declared with such "holes" when the NDL program is compiled, or a line may be reconfigured into such a state by an MCS. In some cases, a line may exist with no stations at all. At no time, however, may more stations be assigned to a line than the maximum number defined by the NDL program. Thus, it is the requirement of the DCP operating system and/or the **CONTROL** definitions to ensure that a selected station actually exists, or is valid, as defined by the current state of the network.

Just as a line may have no valid stations, it is possible and often desirable to define "spare" stations which have no line assignment. Such stations cannot be referenced or utilized by the DCP until they are logically assigned to a line by a reconfiguration request. Again, any such reconfiguration request will be disallowed by the DCC if the characteristics of the station conflict with those of the line to which it is being assigned, or with the stations which already exist on that line. Also, since the size of the line table cannot be altered, there must exist a "hole" or unused position on the line for the station.

Alternatively, an existing station may be subtracted from a line, thereby leaving a "hole," and either left in limbo with no line assignment, or moved to fill an existing "hole" on another line. Thus, stations which have special characteristics for a particular application may be logically moved about within the network while the data communications system is operating and without the further use of the NDL compiler.

DCP MESSAGE MAINTENANCE

With one exception, all functions performed by the DCP are the direct result of a DCP request message being sent to the DCP, usually by an MCS. In the case of terminal output, for example, a "write request" is sent to the DCP, which then invokes the action described within the request message itself by means of the appropriate station's **TRANSMIT REQUEST** definition. If spontaneous input from a terminal is to be received, there is normally no MCS request message associated with the input operation. When a station operates in this mode, the terminal is described as being "enabled for input," or simply, "enabled."

The process of gathering "enabled input messages," i.e., spontaneous input messages, is controlled by the **CONTROL** definition, and, of course, by the **RECEIVE REQUEST** defined for the terminal. In addition, the "enabled" state of a station is initially defined for each station, and may be dynamically changed by the controlling MCS.

When a station is enabled, and the **RECEIVE REQUEST** is invoked, the DCP must then acquire a message area in main memory in which to store the received message text. Such an area, which is called an "enable input space," is obtained by a DCP macro called **GETSPACE**. Since the DCP cannot directly participate in the main system memory management functions, a pool of such "enable input" spaces is maintained by the DCC. This pool of messages, sometimes referred to as the "available space pool," consists of a set of queues, each of which contains a linked list of available message areas of the same size. The NDL compiler computes the size of the enable input space required for each terminal based on the defined **WIDTH**, **MAXINPUT**, and **BUFFERSIZE** statements within the terminal definition. All terminals of a given size are assigned the use of the same queue. In order to reduce the number of different queues required, the NDL compiler rounds each terminal's input size up to a multiple of 16 words.

The available space areas are used for several purposes other than terminal enabled input. Error messages from the DCP and "switched line status" result messages are also spontaneous in nature and require an enable input space. In addition, it is possible for an NDL Request definition to invoke the **GETSPACE** macro and simply store the contents of variables in the obtained message space in order to communicate with the controlling MCS.

When the DCP **GETSPACE** macro is invoked, an area which is greater than or equal to the required message size is delinked from a message queue and assigned to the station. If no suitably sized areas exist within the space pool, a "no space" condition results, and the **RECEIVE REQUEST** must abort reception of input.

The number of messages assigned to each queue is initially defined by the NDL compiler. By default, two areas are assigned to each size queue, although the NDL program may specify an alternate allotment on a terminal by terminal basis.

The DCC has the responsibility of maintaining the available space pool so that **GETSPACE** may always obtain a message area. As each available space area is returned to the DCC by the DCP in the normal course of completing an input operation, the queue from which the area was obtained is restored so that it contains the same number of areas as defined by the NDL compiler. Circumstances may arise, however, where all of the areas within a queue have been exhausted, but none of the areas has yet been returned to the DCC so that the queue can be replenished. In such an event, the DCP sets a global "space alarm" flag which is sensed by the DCC and causes it to immediately examine and replenish all of the available space queues. In addition, the DCC will then increase the target number of messages in each totally depleted queue, in order to reduce the possibility of future space alarms. During extended periods of DCP inactivity, the DCC will attempt to reduce the number of messages in each queue down toward the originally defined target value.

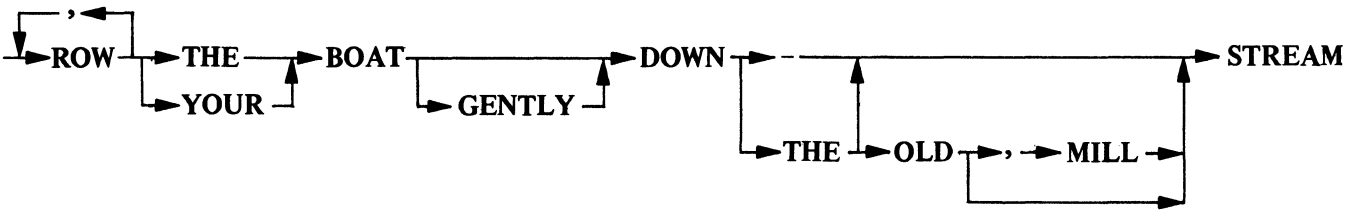
The DCC attempts to maintain the available space pool within the constraints specified by the NDL compiler. However, some networks may require more than the default number of message areas for some terminals if too many "no space" conditions occur. In such an event, the NDL program should specify a larger number of message areas for the affected terminals. Since the behavior of a network is difficult to predict under all circumstances, the NDL programmer will have to directly observe the effects of different message space specifications, and adjust the specifications so that the network operates efficiently without requiring excessive memory resources.

CHAPTER 2

NDL SYNTAX CONVENTIONS

SYNTAX CONVENTIONS

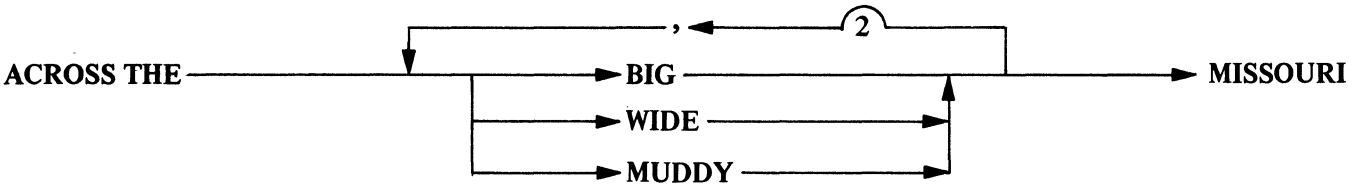
The syntax diagram is the method used to depict the Network Definition Language syntax. This method affords a very concise and lucid exposition of syntax, including defaults, alternatives, and iterations; it is rigorous without being cumbersome. There are few formal rules to remember: the basic rule is that any path traced along the forward directions of the arrows produces a syntactically valid command. The following examples illustrate the technique:



Valid constructs from this syntax diagram include:

ROW THE BOAT DOWN-STREAM.
ROW, ROW, ROW YOUR BOAT GENTLY DOWN THE STREAM.
ROW, ROW, ROW, ROW THE BOAT DOWN THE OLD STREAM.
ROW YOUR BOAT DOWN THE OLD, MILL STREAM.
ROW THE BOAT DOWN THE OLD, MILL STREAM.

The following convention is used to control iterations of options or constructs:



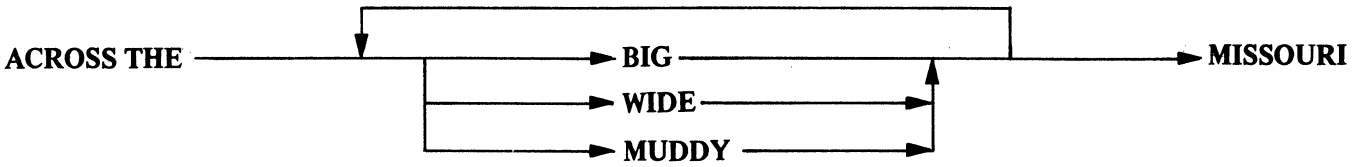
The "bridge" over the "2" can be crossed a maximum of two times, so a maximum of two commas (and three adjectives) can appear. Valid productions include:

ACROSS THE BIG MISSOURI.
ACROSS THE BIG, WIDE MISSOURI.
ACROSS THE BIG, WIDE, MUDDY MISSOURI.

From the above example, it should be noted that the number of iterations is controlled by the "number" in the feed-back loop. When the "number" is not shown, there is no limit to the iterations. For example,

SYNTAX CONVENTIONS

Continued



would include the following valid combinations:

ACROSS THE BIG BIG WIDE WIDE MISSOURI.
ACROSS THE BIG MISSOURI.
ACROSS THE MUDDY MUDDY MUDDY MUDDY MISSOURI.

If a comma were included in the above example, valid combinations would be as follows:

ACROSS THE BIG, BIG, MUDDY MISSOURI.
ACROSS THE BIG, WIDE, WIDE, MUDDY MISSOURI.

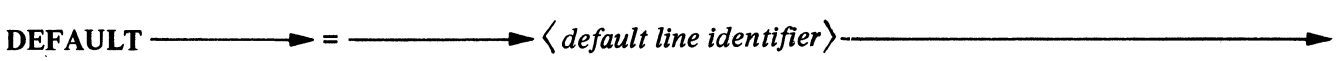
Key Words

Boldface symbols and uppercase letters in syntax diagrams indicate symbols and words which appear literally in the instruction.

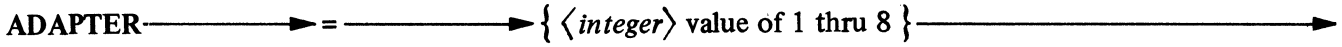
Syntactic Variables

In the syntax diagrams, left and right broken brackets (⟨ ⟩) are used to contain syntactic variables that represent information to be supplied by the programmer. A particular variable may represent a single character, a simple construct (such as an integer or text string), or a relatively complicated construct.

The following is an example of a syntactic variable that appears in a syntax diagram.



Braces ({ }) are used to enclose syntactic variable expressions defined by the meaning of the English language expression contained within the braces. For example, the following syntactic variable expression



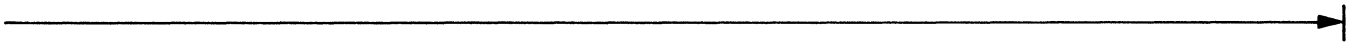
would include the following valid constructs:

ADAPTER = 1.
ADAPTER = 6.
ADAPTER = 8.

Construct Terminator

Most constructs in the Network Definition Language must be terminated by a period (.) as indicated in the syntax examples.

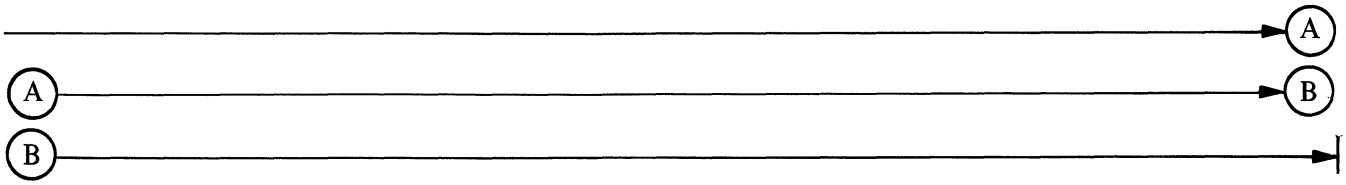
Some constructs, however, do not require a terminator, and can be followed by another construct. This is illustrated as follows:



The vertical bar (|) is not part of the syntax, but merely indicates the termination of the construct.

CONSTRUCT CONTINUATION

Some syntax diagrams are too large to fit on a single line, in this manual. This condition requires that the diagram must be continued, and such a continuation is represented by the use of circled letter terminators, as follows:



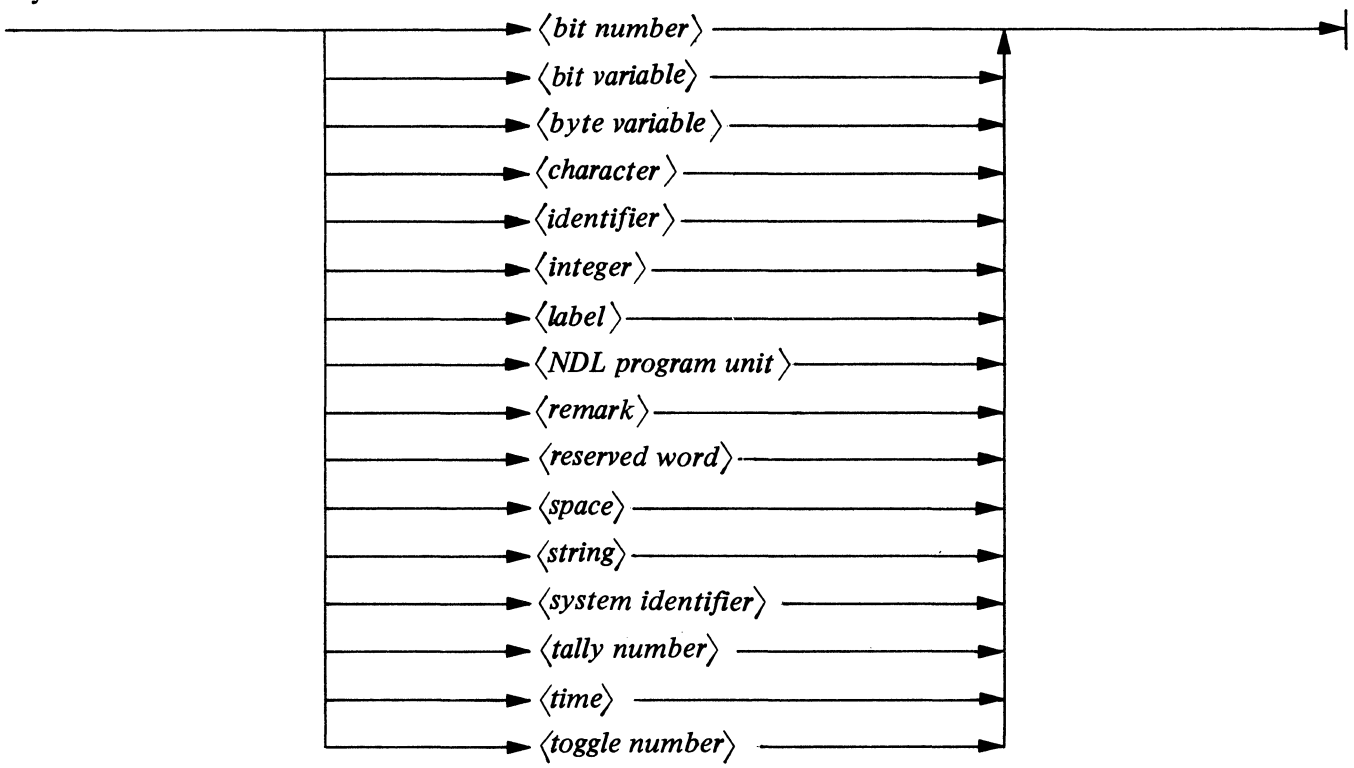
The circled letter symbol is not part of the syntax, but merely shows that the diagram is continued. The syntax diagram is resumed on the subsequent syntax line that begins with a duplicate of the circled letter continuation symbol.

CHAPTER 3

LANGUAGE COMPONENTS

LANGUAGE COMPONENTS

Syntax



Examples

```
A
450
6110
IF
"B6700"
SYSTEM/CANDE
30 MILLI
```

Semantics

⟨*bit variable*⟩*s*, ⟨*bit number*⟩*s*, and ⟨*byte variable*⟩*s* are all described in chapter 6.

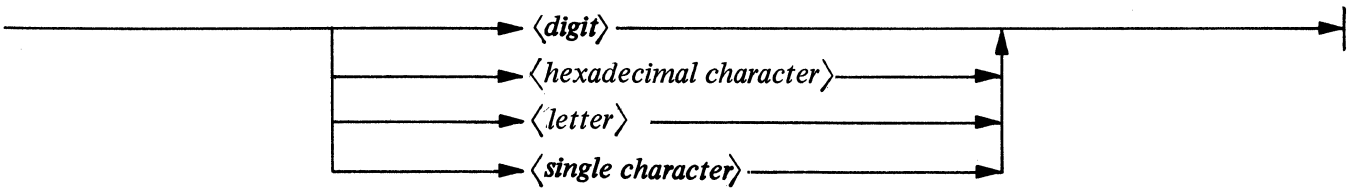
A complete list of the ⟨*reserved word*⟩*s* is contained in appendix A.

⟨*NDL program unit*⟩ is described in chapter 4.

All other ⟨*language component*⟩*s* are described in this chapter.

CHARACTER

Syntax



Examples

```
O
Q
A
"Z"
```

Semantics

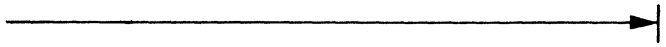
In all instances, a ⟨*character*⟩ is an entity whose exact form depends on the context of its usage. The normal inference is that of an 8-bit EBCDIC character.

⟨*letter*⟩*s* and ⟨*digit*⟩*s* are usually used to create ⟨*identifier*⟩*s* and ⟨*string*⟩*s*.

Wherever ⟨*single character*⟩ occurs in the NDL syntax, an 8-bit character is needed. It is unique in that it may be formed using two ⟨*hexadecimal character*⟩*s*.

DIGIT

Syntax

{one of the EBCDIC characters, 0 through 9, inclusive} 

Examples

0
5
9

Semantics

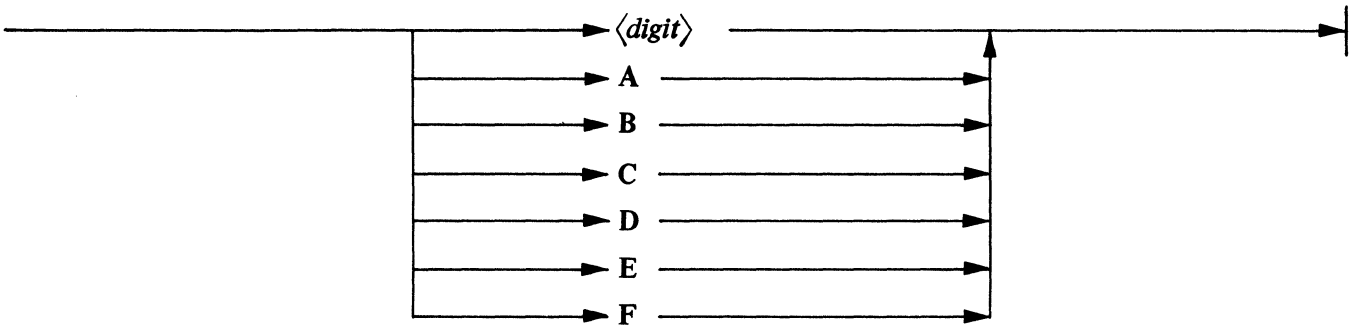
Whenever the item of ⟨*digit*⟩ appears in the NDL syntax, one of the 10 numeric EBCDIC characters from 0 through 9 is required.

CHARACTER

⟨hexadecimal character⟩

HEXADECIMAL CHARACTER

Syntax



Examples

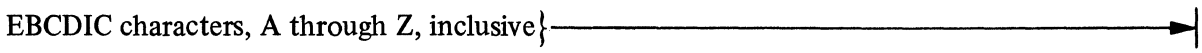
0
5
9
A
C
F

Semantics

⟨hexadecimal character⟩s are defined as consisting of the characters in the decimal digit set plus the characters A, B, C, D, E, F. *⟨hexadecimal character⟩s* are generally used to define program values in terms of the hexadecimal (radix 16) number system, where A is equivalent to 10 in the decimal system, B is equivalent to 11 in the decimal system, etc.

LETTER

Syntax



Examples

A
Q
Z

Semantics

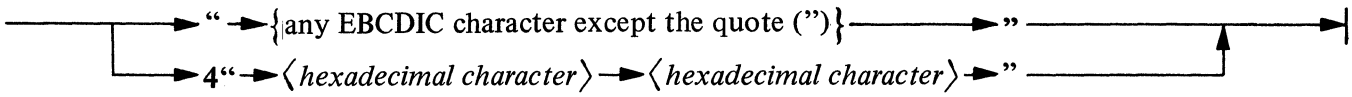
Whenever the item of ⟨*letter*⟩ appears in the NDL syntax, one of the 26 alphabetic EBCDIC characters from A through Z is required.

CHARACTER

⟨single character⟩

SINGLE CHARACTER

Syntax



Examples

"A"
4"FF"

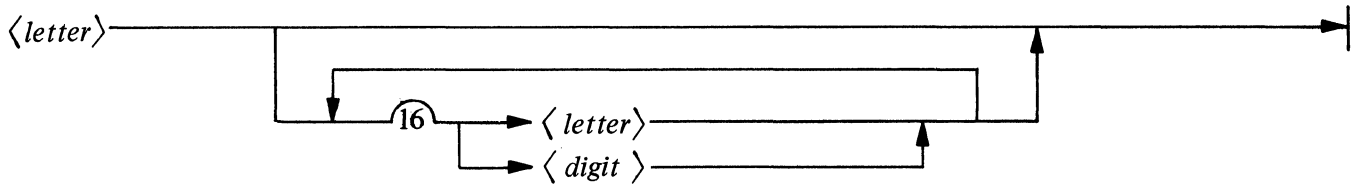
Semantics

The primary purpose of having a syntactical item of *⟨single character⟩* in NDL is for use in those places of syntax requiring an 8-bit character, which can be any combination of bits from "all off" through "all on."

For ease of programming and recognition of usage, the NDL programmer may use either normal EBCDIC graphic characters or *⟨hexadecimal character⟩s* to create the needed bit pattern.

IDENTIFIER

Syntax



Examples

A
QV
X3
B6700
MINIMIZER

Semantics

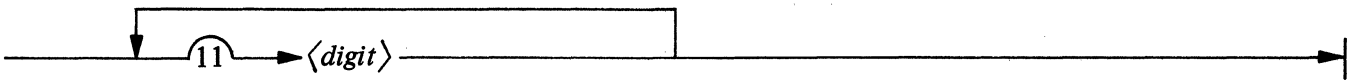
⟨*identifier*⟩*s* have no intrinsic meaning. They are used to give symbolic names to various definitions in NDL.

An ⟨*identifier*⟩ must start with a ⟨*letter*⟩, which can be followed by any combination of ⟨*letter*⟩*s* and ⟨*digit*⟩*s*.

The maximum length of an ⟨*identifier*⟩ is 17 characters.

INTEGER

Syntax



Examples

```
0
37
511
12345678901
```

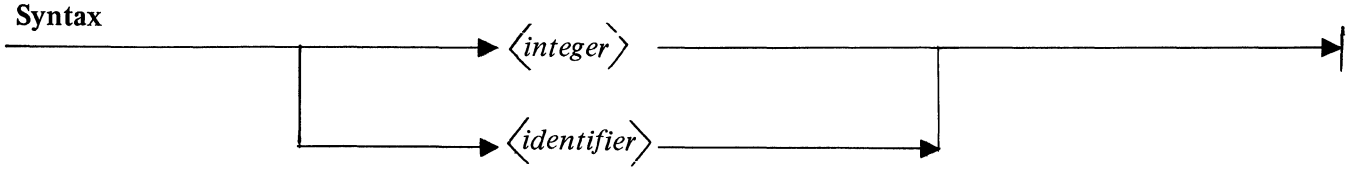
Semantics

An ⟨*integer*⟩ is a positive whole number; i.e., fractions or fractional parts, exponents, etc. are not allowed.

The maximum ⟨*integer*⟩ allowed is 99999999999.

LABEL

Syntax



Examples

```
0
99999999999
STORE CHAR
SKIPLINES
```

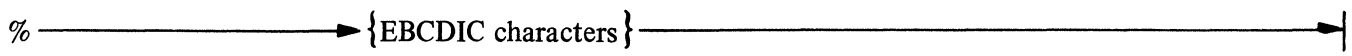
Semantics

A ⟨*label*⟩ is used to indicate where "execution can branch" within a given ⟨*control definition*⟩ or ⟨*request definition*⟩.

⟨*label*⟩*s* are "local" in scope; that is, each ⟨*label*⟩ must be unique only within a given ⟨*control definition*⟩ or ⟨*request definition*⟩. For example, the ⟨*label*⟩ 22 could appear more than once in an NDL program so long as it does not appear more than once in the same ⟨*control definition*⟩ or ⟨*request definition*⟩.

REMARK

Syntax



Example

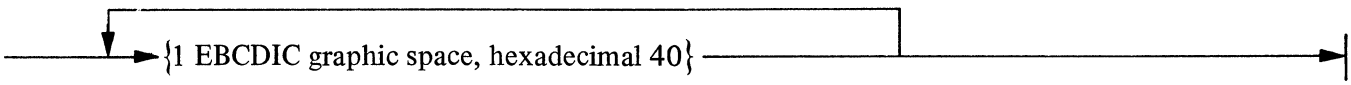
```
% THIS IS A REMARK
```

Semantics

⟨remark⟩s can appear anywhere in the source program. When the compiler encounters the percent sign (%), it skips immediately to the next source record before continuing the compilation process.

SPACE

Syntax



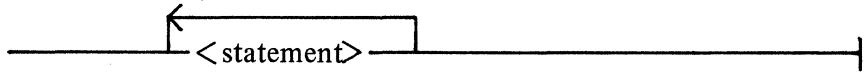
Semantics

The NDL compiler looks at a contiguous sequence of ⟨*space*⟩ *s* in a source program as a single ⟨*space*⟩ (except when contained in a ⟨*string*⟩). Therefore, wherever a single space is allowed, the programmer can use multiple ⟨*space*⟩ *s* to improve readability of the program.

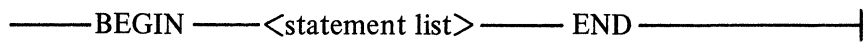
STATEMENT

Syntax

<statement list>



<compound statement>



Examples <statement list>

```
AI=CHARACTER ASR3.
CODE = ASCII.
FETCH SEQUENCE [ENDOFSEQ].
FINISH TRANSMIT.
```

Example <compound statement>

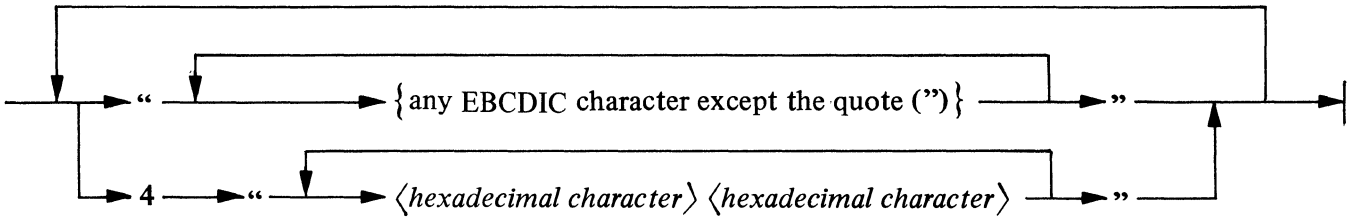
```
BEGIN                          %EXAMPLE 1
INITIATE TRANSMIT.
TRANSMIT EOT.
FINISH TRANSMIT.
END.

IF STATION (VALID) THEN
  IF STATION (READY) THEN
      BEGIN                    %EXAMPLE 2
      INITIATE TRANSMIT.
      TRANSMIT "ON THE AIR".
      FINISH TRANSMIT.
      END.
```

Semantics

An NDL statement is defined as one or more ⟨statement⟩ s in a ⟨statement list⟩ or a ⟨compound statement⟩ that groups a ⟨statement list⟩ into a logical unit.

STRING

Syntax



Examples

"THIS IS A STRING"
"AND" "SO" "IS" "THIS"
"AND SO" 4"C9E2" "THIS"
4"C2F6F7F0F0"
"%*+?/@ (BIG B)"

Semantics

Only ⟨*hexadecimal character*⟩ (4-bit) and EBCDIC character (8-bit) ⟨*string*⟩*s* can be constructed.

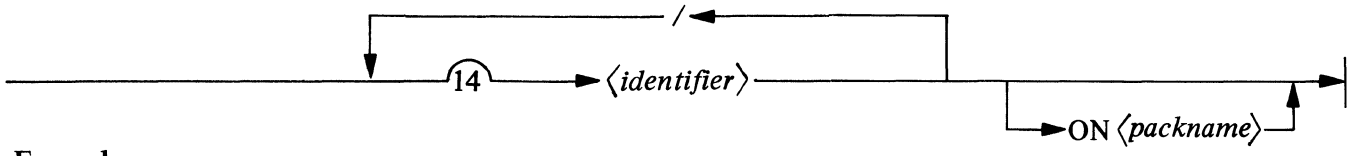
Hexadecimal character strings are restricted in that they cannot contain the character string 4 "7F".

EBCDIC character ⟨*string*⟩*s* are restricted in that they cannot contain the quote (") character.

The maximum allowed length of a ⟨*string*⟩ is 128 8-bit characters (1024 bits).

SYSTEM IDENTIFIER

Syntax



Examples

```
A
B6700
SITE/MCS
SYSTEM/CANDE ON PACK
SYSTEM/RJE/DOWNTOWN
SYSTEM/RJE ON SIXPACK
X/Y/ZEBRA
```

Semantics

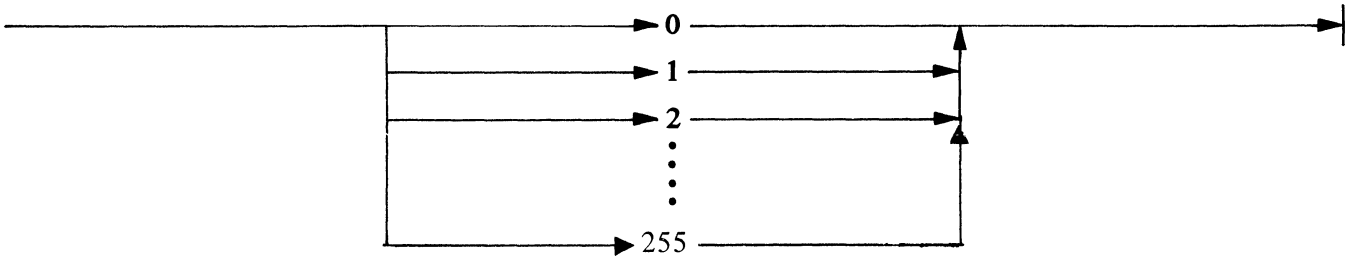
⟨*system identifier*⟩*s* have no intrinsic meaning. They are used to give symbolic names to various definitions in NDL. A ⟨*system identifier*⟩ is different from an ⟨*identifier*⟩ in that it is usually used to reference items belonging to the system and not simply to the NDL program.

A maximum of 14 ⟨*identifier*⟩*s*, each separated by a slash (/), is allowed.

ON ⟨*packname*⟩ is valid only for MCS names; not for station or file identifiers.

TALLY NUMBER

Syntax



Examples

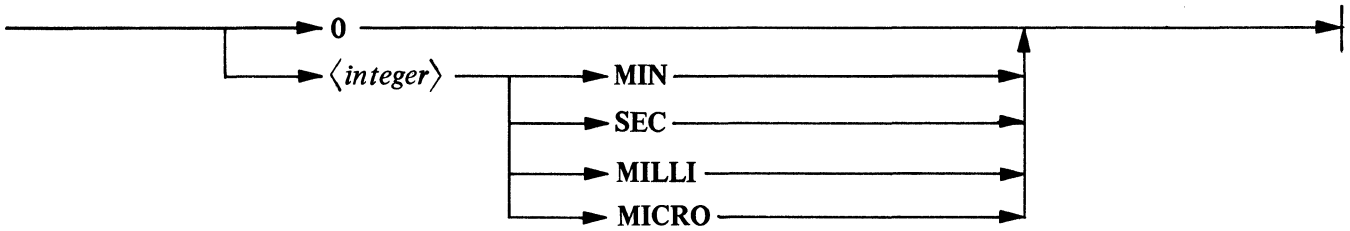
```
0
1
2
```

Semantics

⟨*tally number*⟩ is required to designate one of the 256 ⟨*byte variable*⟩ **TALLY**s; for example, **TALLY[0]**.

TIME

Syntax



Examples

```
0
5 MIN
30 SEC
200 MILLI
9 MICRO
```

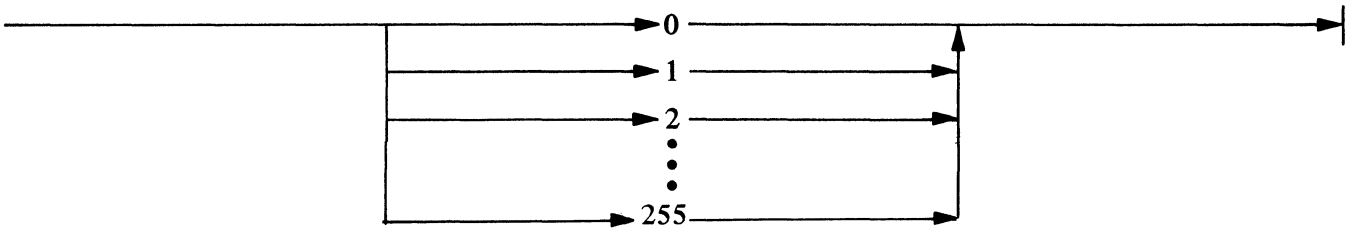
Semantics

⟨*time*⟩ is used to express or define an increment of time. **MIN** denotes minutes, **SEC** denotes seconds, **MILLI** denotes milliseconds, and **MICRO** denotes microseconds.

The maximum amount of time that can be specified is 6 minutes 42 seconds.

TOGGLE NUMBER

Syntax



Examples

0
4
7

Semantics

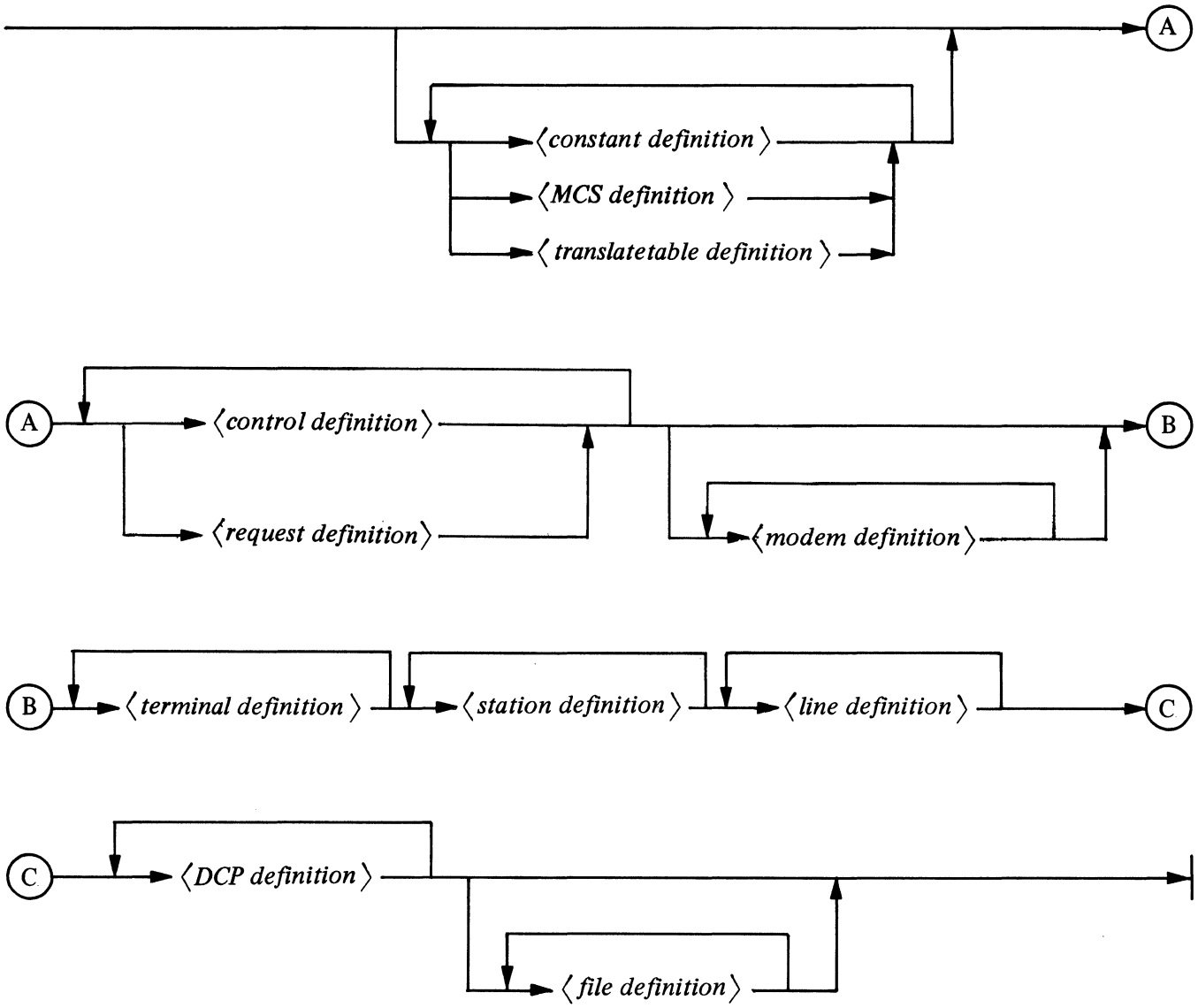
⟨ *toggle number*⟩ is required to designate one of the 256 ⟨*bit variable*⟩ **TOG**s. For example, **TOG[5]**.

CHAPTER 4

SOURCE PROGRAM STRUCTURE

NDL PROGRAM UNIT

Syntax



NDL PROGRAM UNIT

Continued

Examples

```
CONSTANT . . .
MCS . . .
TRANSLATETABLE . . .
CONTROL . . .
REQUEST . . .
MODEM . . .
TERMINAL . . .
STATION . . .
LINE . . .
DCP . . .
FILE . . .
```

Semantics

The NDL source program is divided into 11 program sections ordered as shown. An NDL program must include the control and request sections (in any order), and the terminal, station, line, and DCP sections. Each section is described in detail in chapter 5 of this manual.

CHAPTER 5

DEFINITIONS

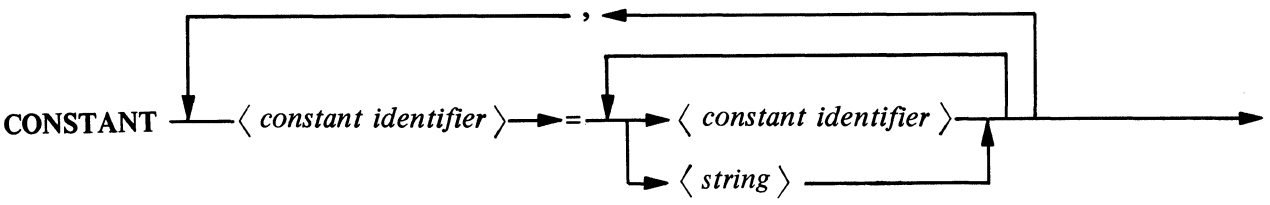
DEFINITIONS

The NDL definitions which comprise the 11 program sections of the source program structure shown in Section 4 and the **DEFINE** definition which is global to the **CONTROL** and **REQUEST** definitions are listed in alphabetical order and described in the same order:

a. CONSTANT
b. CONTROL
c. DCP
d. DEFINE
e. FILE
f. LINE
g. MCS
h. MODEM
i. REQUEST
j. STATION
k. TERMINAL
l. TRANSLATETABLE

CONSTANT DEFINITION

Syntax



Examples

```
CONSTANT   NUL = 4"00".
CONSTANT   SOH = 4"01", STX = 4"02".
CONSTANT   C1  = SOH 4"00" STX, C2 = "123"4"F4".
```

Semantics

The ⟨*constant definition*⟩ equates each of one or more ⟨*identifier*⟩*s* with a ⟨*string*⟩. Once that equation is made, any subsequent appearance of the ⟨*constant identifier*⟩ is syntactically and semantically equivalent to the ⟨*string*⟩.

If a ⟨*constant identifier*⟩ appears after the equal sign, it must have been defined previously in the program.

Supplementary Examples

Examples of Valid ⟨*constant definition*⟩*s*

Example 1

```
CONSTANT GREETING = "WELCOME TO B 6700 TIME SHARING.".
```

This example equates the ⟨*string*⟩ "WELCOME TO B 6700 TIME SHARING." to the ⟨*constant identifier*⟩ GREETING.

Example 2

```
CONSTANT

    CR          = 4"0D", % A CARRIAGE RETURN
    LF          = 4"25", % A LINE FEED
    CRANDLF     = CR LF, % A CARRIAGE RETURN AND A LINE FEED
    DELETELINE = "DELETED" CR LF.  % THE STRING "DELETED",
                                   % A CARRIAGE RETURN, AND
                                   % A LINE FEED.
```

This example references other ⟨*constant identifiers*⟩*s* to define a ⟨*constant identifier*⟩. ⟨*String*⟩*s* and ⟨*constant identifier*⟩*s* may be interspersed to define a ⟨*constant identifier*⟩.

Examples of Invalid ⟨*constant definition*⟩*s*

Example 1

```
CONSTANT BADCNST = 4"123".
```

The above ⟨*constant definition*⟩ would cause a syntax error to be generated, because the ⟨*string*⟩ is not properly formed. The length of the ⟨*string*⟩ must be a multiple of eight bits.

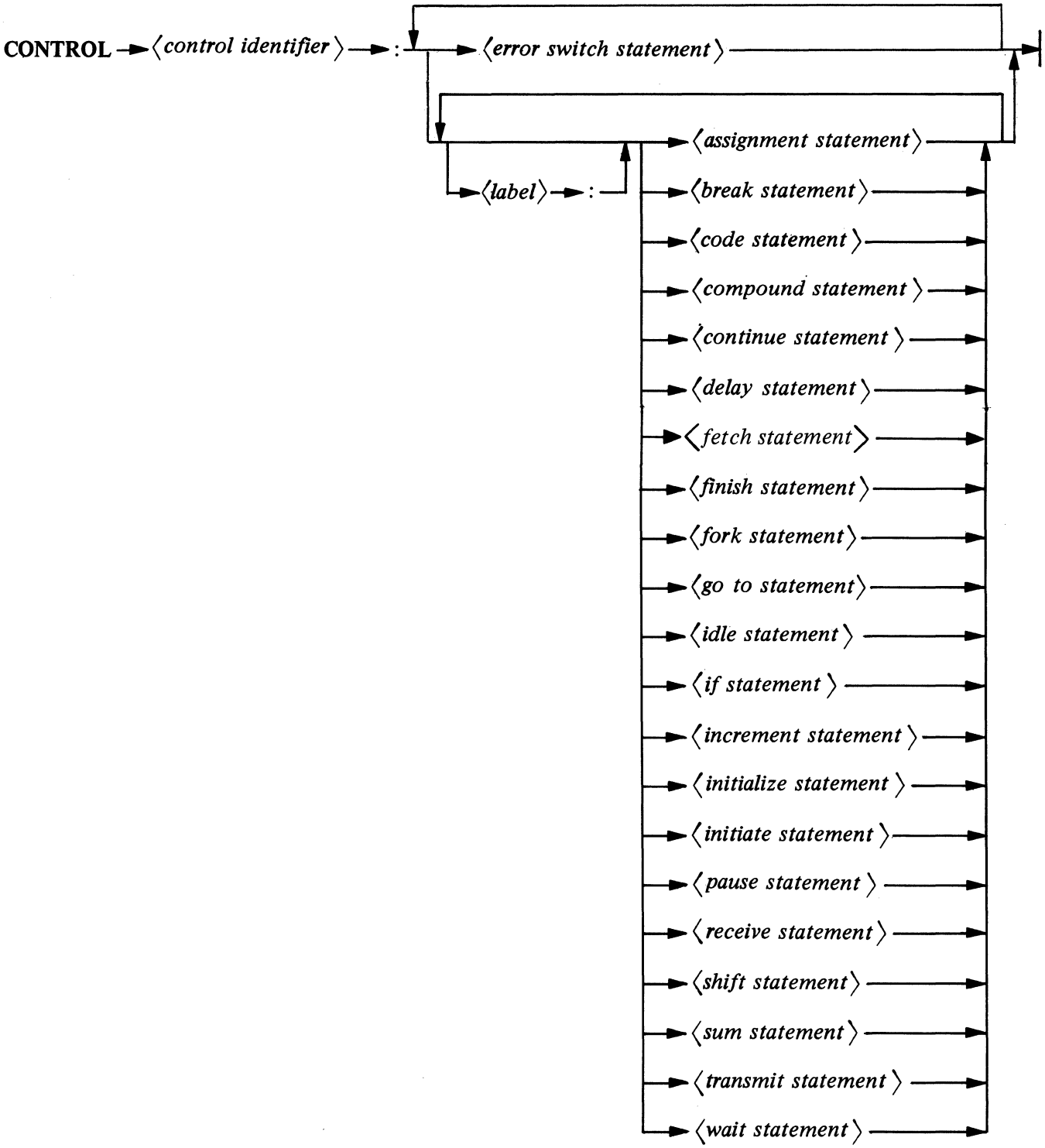
Example 2

```
CRANDLF = CR LF,   % LINE 1
CR      = 4"0D",   % LINE 2
LF      = 4"25".   % LINE 3
```

This example would cause a syntax error to be generated, because the ⟨*constant identifier*⟩*s* **LF** and **CR** are referred to in line 1, but not declared until line 2 and line 3. A ⟨*constant identifier*⟩ must be declared before any reference to that ⟨*constant identifier*⟩ can be made.

CONTROL DEFINITION

Syntax



Examples

```
CONTROL CONTENTION:
    INITIATE REQUEST.
    INITIATE ENABLEINPUT.
    IDLE.

CONTROL POLL:
    5:  IF STATION GTR 0 THEN
            BEGIN
    10:     STATION = STATION – 1.
            INITIATE REQUEST.
            INITIATE ENABLEINPUT.
            END

        ELSE
            BEGIN
            STATION = MAXSTATIONS.
            GO TO 10.
            END.
        GO TO 5.
```

Semantics

A ⟨*control definition*⟩ is an algorithm that describes the allocation of the use of a logical line to the stations assigned to that line. It is the ⟨*control definition*⟩ that decides if and when a station's Receive Request or Transmit Request should be initiated.

A single ⟨*control definition*⟩ must control the logical line resource for all of the stations on a half-duplex line. In the case of full duplex, one ⟨*control definition*⟩ must control the primary line, and one additional ⟨*control definition*⟩ can be designated to control the auxiliary line. (One ⟨*control definition*⟩ could be designated as the control for both the primary and auxiliary lines. If a ⟨*control definition*⟩ is not designated as the control for an auxiliary line, then a default equivalent to an ⟨*idle statement*⟩ is used.) The programmer, however, cannot directly define a particular ⟨*control definition*⟩ for a logical line in its ⟨*line definition*⟩. Instead, for each ⟨*terminal definition*⟩, a single ⟨*control definition*⟩ must be defined. (Two ⟨*control definition*⟩*s* can be named if the ⟨*terminal duplex statement*⟩ specifies **DUPLEX=TRUE.**) Next, in each ⟨*station definition*⟩, a ⟨*terminal definition*⟩ must be defined for the station (by means of the ⟨*station terminal statement*⟩). Finally, each ⟨*station definition*⟩ is assigned to a logical line (in the ⟨*line station statement*⟩ of the ⟨*line definition*⟩). This last association must be such that each station (i.e., ⟨*station definition*⟩) assigned to the logical line references (indirectly through its ⟨*terminal definition*⟩) the same ⟨*control definition*⟩*s* as every other station assigned to the line.

A ⟨*control definition*⟩ for a given line can deal only with one station at a time. All statements executed apply to and affect only one station assigned to the line. The ⟨*control definition*⟩ chooses the station with which it wishes to deal by setting the value of the ⟨*byte variable*⟩ STATION to the chosen station's station index.

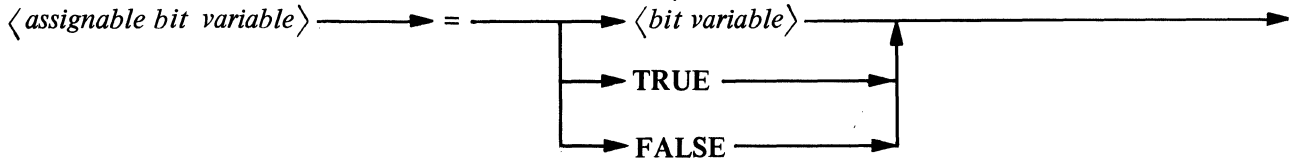
⟨*control identifier*⟩ has the same syntactic form as ⟨*identifier*⟩.

The statements in a ⟨*control definition*⟩ are executed sequentially. In some cases, however, it is desirable to alter the order of execution of statements within the procedure. A ⟨*control statement*⟩ preceded by a ⟨*label*⟩ is one means of accomplishing this. The ⟨*go to statement*⟩ is used to transfer control to a ⟨*label*⟩*ed* ⟨*control statement*⟩.

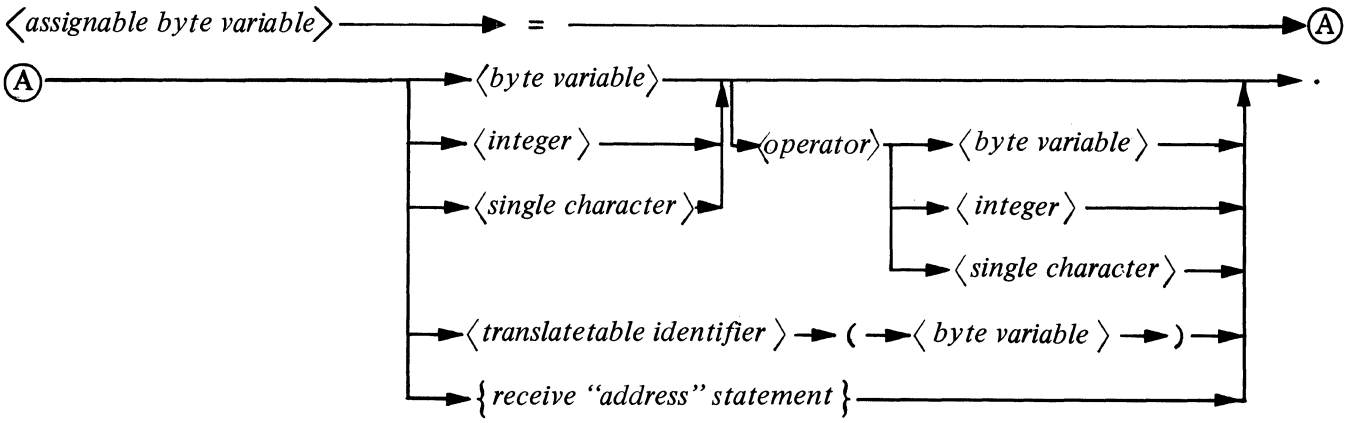
ASSIGNMENT STATEMENT

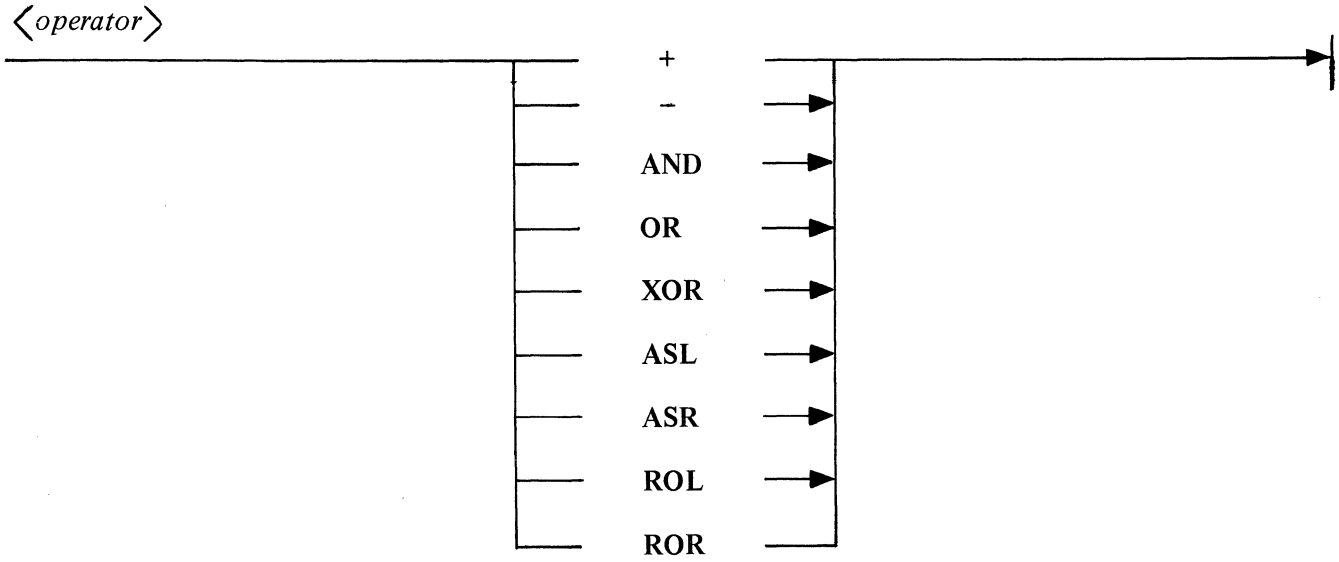
Syntax

FORM 1 - LOGICAL ASSIGNMENT



FORM 2 - VALUE ASSIGNMENT





Examples:

```
TOG[0] = TRUE.
TOG[1] = TOG[0].
LINE (BUSY) = FALSE.
STATION = MAXSTATIONS.
```

```
TALLY[0] = STATION(FREQUENCY) - TALLY[1].
CHARACTER = TRANSTABLEID (CHARACTER).
STATION = RECEIVE ADDRESS (TRANSMIT) [ADDERR: 999].
AI = CHARACTER ASR 3.
TALLY[2] = TALLY[0] ASL 1.
LINE(TALLY[1]) = STATION(FREQUENCY) ROL RETRY.
CHARACTER = CHARACTER ROR 6.
TALLY[0] = CHARACTER AND 4 "03".
```

Semantics

FORM 1

This form causes the value on the right side of the equal sign to replace the current value of ⟨*assignable bit variable*⟩.

FORM 2

Value assignment causes a calculated value on the right of the equal sign to be stored in the ⟨*assignable byte variable*⟩.

Arithmetic calculations (+ and -) are done in modulo 256 arithmetic.

The logical operators **AND**, **OR**, and **XOR** are defined by table 5-0.

The logical operations defined above are performed on all 8 bits of the operands (⟨*byte variables*⟩s, ⟨*integers*⟩s, and/or ⟨*single characters*⟩s).

The shift operators (**ASL**, **ASR**, **ROL**, **ROR**) allow for "shifting" the contents of byte variables or constants.

The **ASL** and **ASR** operators are arithmetic shifts, to the left and right, respectively. The first operand is shifted by a number of bits specified by the second operand, with zeroes being shifted into the operand and bits being lost from the opposite end.

For example, if

```
TALLY[0] = 4.
```

then

```
TALLY[2] = TALLY[0] ASL 1.
```

would double the value of **TALLY[0]** by shifting it left one, thus storing the value, 8, in **TALLY[2]**.

TABLE 5-0. TRUTH TABLE

BIT A	BIT B	A AND B	A OR B	A XOR B
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	FALSE

Similarly, if

CHARACTER = 4 "3C".

then

AI = CHARACTER ASR 3.

would shift the value 4 "3C" right by 3 bits, resulting in the value, 4 "07" being placed in the AI register.

The **ROL** and **ROR** operators are rotational shifts, with bits being shifted back into the operand from which they are lost.

Thus if

STATION(FREQUENCY) = 9.
RETRY = 6.

then

LINE(TALLY[1]) = STATION(FREQUENCY) ROL RETRY.

would cause the contents of **STATION(FREQUENCY)**, 9, to be rotated left 6 bits, the value of **RETRY**. Thus the value 4 "42" would be stored in **LINE(TALLY[1])**.

Likewise, if

CHARACTER = 4 "3C".

then

CHARACTER = CHARACTER ROR 6

would rotate the contents of **CHARACTER** right six bits. The resulting value, 4 "F0", is then stored back in **CHARACTER**.

⟨*assignable byte variable*⟩ = ⟨*translatetable identifier*⟩ (⟨*byte variable*⟩).

This construct is the means to invoke user-defined character translation. User-defined translation is effected by three areas of the NDL source program.

a. In a ⟨*translatetable definition*⟩ the programmer must define the contents of a translation table and associate a ⟨*translatetable identifier*⟩ with it.

b. In the ⟨*terminal definition*⟩ of a terminal type that requires special character translation the programmer should suppress automatic character translation by using either of the following forms of ⟨*terminal code statement*⟩:

CODE = BINARY.

or

CODE = EBCDIC.

c. In a ⟨*control definition*⟩ or ⟨*request definition*⟩, the programmer invokes the translation by using this option of the value assignment. Any ⟨*byte variable*⟩ can be designated as containing the character to be translated.

The ⟨*translatetable identifier*⟩ identifies the translation table to be used. An ⟨*assignable byte variable*⟩ is designated to the left of the equal sign, identifying where the resulting translated character is to be stored.

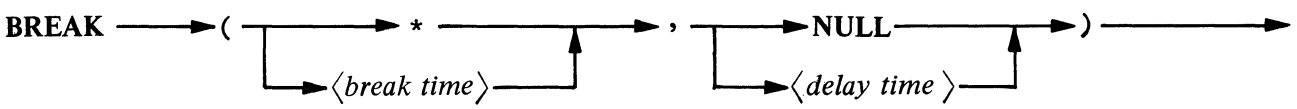
If N is the ⟨*source size*⟩ (defined in the ⟨*translatetable definition*⟩), then the N low-order bits of the ⟨*byte variable*⟩ are used as an index into the translation table. The eight-bit character thus indexed is stored in the ⟨*assignable byte variable*⟩.

⟨*assignable byte variable*⟩ = {*receive "address" statement*}.

This construct attempts to **RECEIVE** the address characters of a terminal, and store in ⟨*assignable byte variable*⟩ the station index of a station whose address characters are equal to those received. The {*receive "address" statement*} is the same as described in the semantics of the **RECEIVE ADDRESS** option of the ⟨*receive statement*⟩. The optional syntax in the {*receive "address" statement*} invokes the same actions as described in the ⟨*receive statement*⟩ semantics except for **ADDERR.** Any action specified for **ADDERR** is taken if no valid station assigned to the line is found with address characters equal to those received.

BREAK STATEMENT

Syntax



Examples

```
BREAK  (*, NULL).
BREAK  (200 MILLI, 3 SEC).
BREAK  (*, 3 SEC).
BREAK  (100 MILLI, NULL).
```

Semantics

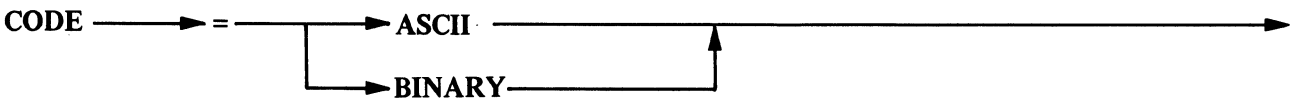
The ⟨*break statement*⟩ causes binary zeros to be transmitted on the line, thus changing the state of the line to a "spacing" condition for a specified time.

⟨*break time*⟩ specifies the ⟨*time*⟩ to break. An asterisk indicates that a standard break of 2 character times should be used.

⟨*delay time*⟩ specifies the ⟨*time*⟩ to delay subsequent to the break and prior to when control continues.

CODE STATEMENT

Syntax



Semantics

CODE=ASCII invokes the ASCII-to-EBCDIC translation for received data and the EBCDIC-to-ASCII translation for transmitted data.

CODE=BINARY inhibits any character translation on data transmitted or received.

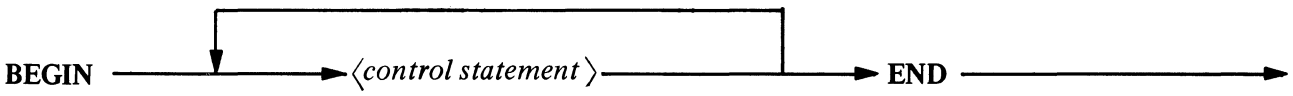
Pragmatics

The ⟨*code statement*⟩ allows a programmer to either invoke or inhibit on a logical line the DCP ASCII-to-EBCDIC character code translation for input, and the EBCDIC-to-ASCII character code translation for output. Any ⟨*terminal definition*⟩ that names, in its ⟨*terminal control statement*⟩, a ⟨*control definition*⟩ that utilizes the ⟨*code statement*⟩, must define **ASCII(BINARY)** as its character code in the ⟨*terminal code statement*⟩. (Refer to the ⟨*terminal code statement*⟩ in this chapter.)

Once that translation has been invoked on a logical line, the translation continues until such time that it is inhibited. If translation is inhibited, translation will be inhibited on that line until invoked again by execution of **CODE = ASCII** or if control is transferred to a ⟨*request definition*⟩ which executes one of the following: **CODE=ASCII**

COMPOUND STATEMENT

Syntax



Examples

```
BEGIN                          % EXAMPLE 1
INITIATE TRANSMIT.
TRANSMIT EOT.
FINISH TRANSMIT.
END.
```

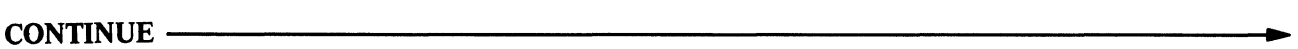
```
IF STATION (VALID) THEN
    IF STATION (READY) THEN
        BEGIN                          % EXAMPLE 2
        INITIATE TRANSMIT.
        TRANSMIT "ON THE AIR".
        FINISH TRANSMIT.
        END.
```

Semantics

The ⟨*compound statement*⟩ groups several statements together to form a logical sequence. To execute more than one statement when the condition of an ⟨*if statement*⟩ is satisfied, a ⟨*compound statement*⟩ must be used.

CONTINUE STATEMENT

Syntax



Semantics

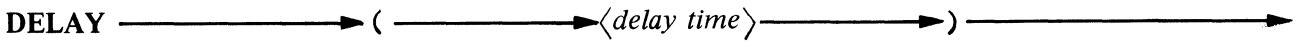
The ⟨*continue statement*⟩ can appear in only those ⟨*control definition*⟩*s* and ⟨*request definition*⟩*s* written to communicate with full duplex terminal types. This statement causes the co-line to resume processing, if, and only if, it had been suspended by a ⟨*wait statement*⟩ or a ⟨*receive statement*⟩ with a **CONTINUE** option specified. If the co-line had not been suspended, this statement acts as a no-op. The ⟨*continue statement*⟩ has no effect upon the line on which it was executed.

Pragmatics

Refer to the ⟨*fork statement*⟩ pragmatics.

DELAY STATEMENT

Syntax



Example

```
DELAY (10 MICRO).
```

Semantics

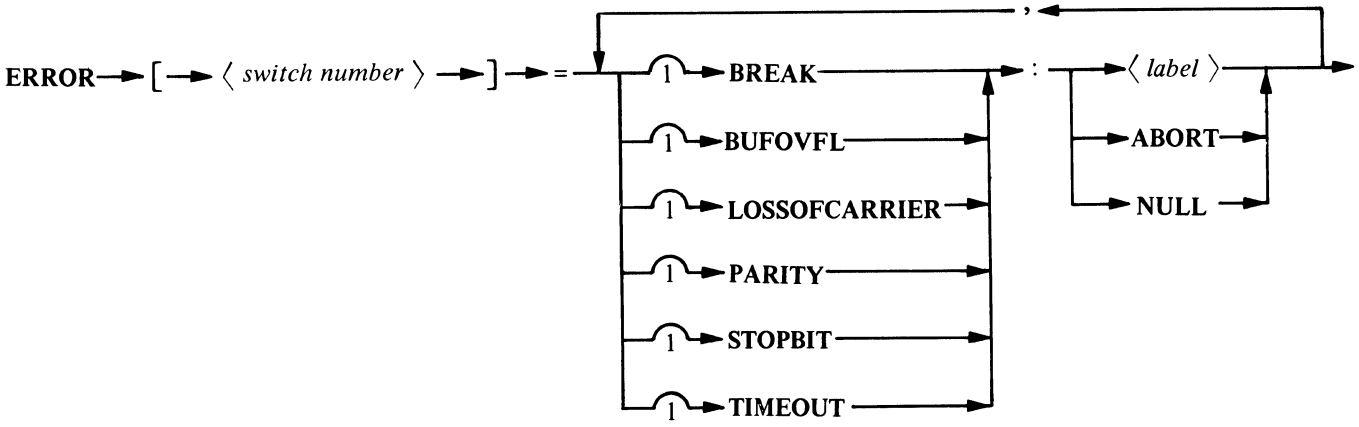
The ⟨*delay statement*⟩ provides a means to delay a specified period of time before control proceeds to the next statement. The ⟨*control definition*⟩ is suspended in a "sleep" state for the ⟨*delay time*⟩ specified.

Pragmatics

The "sleep" state induced by the ⟨*delay statement*⟩ allows the DCP to service Cluster Attention Needed (CAN) interrupts for other logical lines.

ERROR SWITCH STATEMENT

Syntax



Examples

```
ERROR [0]   = BREAK:0,BUFOVFL:NULL,LOSSOFCARRIER:ABORT
    PARITY:999, STOPBIT:999, TIMEOUT:NULL.
ERROR [1]   = BREAK:NULL.
ERROR [99]  = BUFOVFL:NULL .
```

Semantics

The ⟨*error switch statement*⟩ is a non-executable statement that allows the programmer to define a set of default actions that are to be taken in a ⟨*receive statement*⟩ if the specified errors occur.

⟨*switch number*⟩ is an ⟨*integer*⟩ within a range of 0 to 8,388,607.

BREAK

The **BREAK** option variations cause actions as described if a break, that is, at least 2 character-times of a spacing line condition, is detected by the adapter cluster while receiving:

BREAK:NULL	sets **TRUE** the ⟨*bit variable*⟩ **BREAK[RECEIVE]**. Execution proceeds as if the break did not occur.
BREAK: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **BREAK[RECEIVE]**, and branches control to ⟨*label*⟩.
BREAK:ABORT	sets **TRUE** the ⟨*bit variable*⟩ **BREAK[RECEIVE]**, and executes an implicit **TERMINATE ERROR**.

BUFOVFL

The **BUFOVFL** option variations cause actions as described if the DCP is unable to service a cluster Attention Needed (CAN) interrupt before the Adapter Cluster receives another character (thus destroying the previous character):

BUFOVFL:NULL	sets **TRUE** the ⟨*bit variable*⟩ **BUFOVFL**. Execution proceeds as if the error condition did not occur.
BUFOVFL: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **BUFOVFL**, and branches control to ⟨*label*⟩.
BUFOVFL:ABORT	sets **TRUE** the ⟨*bit variable*⟩ **BUFOVFL**, and executes an implicit **TERMINATE ERROR**.

LOSSOFCARRIER

The **LOSSOFCARRIER** option variations cause actions as described if a loss of carrier is detected while receiving.

LOSSOFCARRIER:NULL	sets **TRUE** the ⟨*bit variable*⟩ **LOSSOFCARRIER**. Execution proceeds as if the error did not occur.
LOSSOFCARRIER: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **LOSSOFCARRIER**, and branches control to ⟨*label*⟩.
LOSSOFCARRIER:ABORT	sets **TRUE** the ⟨*bit variable*⟩ **LOSSOFCARRIER**, and executes an implicit **TERMINATE ERROR**.

There is one exception to the actions described above. If a loss of carrier is detected while receiving, and if the terminal is modem-connect, and if the terminal's ⟨*station definition*⟩ references a ⟨*modem definition*⟩ that contains the statement **LOSSOFCARRIER=DISCONNECT**, then an implicit disconnect is done, regardless of the action associated with **LOSSOFCARRIER** in the ⟨*error switch statement*⟩.

PARITY

The **PARITY** option variations cause actions as described if a parity bit error is detected by the adapter cluster:

PARITY:NULL	sets **TRUE** the ⟨*bit variable*⟩ **PARITY**. Execution proceeds as if the error did not occur.
PARITY: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **PARITY**, and branches control to ⟨*label*⟩.
PARITY:ABORT	sets **TRUE** the ⟨*bit variable*⟩ **PARITY**, and executes a **TERMINATE ERROR.**

STOPBIT

The **STOPBIT** option variations cause the described actions if a stop bit error is detected by the adapter cluster:

STOPBIT:NULL	sets **TRUE** the ⟨*bit variable*⟩ **STOPBIT**. Execution proceeds as if the error did not occur.
STOPBIT: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **STOPBIT**, and branches control to ⟨*label*⟩.
STOPBIT:ABORT	sets **TRUE** the ⟨*bit variable*⟩ **STOPBIT**, and executes a **TERMINATE ERROR.**

TIMEOUT

The **TIMEOUT** option variations cause the actions described if the time required to receive a character exceeds the ⟨*timeout time*⟩. The ⟨*timeout time*⟩ is defined in the ⟨*terminal timeout statement*⟩, but can be overridden by including the (⟨*timeout time*⟩) or (**NULL**) syntax options in the ⟨*receive statement*⟩.

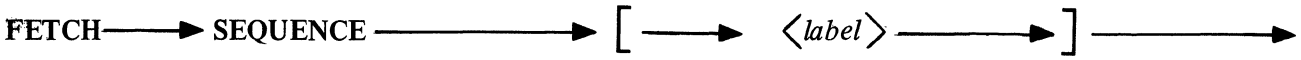
TIMEOUT:NULL	sets **TRUE** the ⟨*bit variable*⟩ **TIMEOUT**. Execution proceeds as if the error did not occur.
TIMEOUT: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **TIMEOUT**, and branches control to ⟨*label*⟩.
TIMEOUT:ABORT	sets **TRUE** the ⟨*bit variable*⟩ **TIMEOUT**, and executes a **TERMINATE ERROR.**

Pragmatics

An ⟨*error switch statement*⟩ must be associated with a ⟨*receive statement*⟩ by means of a ⟨*switch number*⟩ reference before any of the default actions will be invoked. The ⟨*error switch statement*⟩ can appear in a ⟨*control definition*⟩ as many times as the programmer deems convenient providing the following restriction is observed: Within a given ⟨*control definition*⟩, ⟨*error switch statement*⟩*s* must have a unique ⟨*switch number*⟩, and all ⟨*error switch statement*⟩*s* must precede all executable statements in the procedure.

FETCH STATEMENT

Syntax



Semantics

The **FETCH SEQUENCE** statement causes the individual characters which comprise the current sequence number to be loaded into **CHARACTER.** The digits are selected from the most significant to least significant digit. If no more digits remain or if sequence mode is not enabled, a branch to ⟨*label*⟩ is performed.

Thus, the construct:

```
    TRANSMIT SEQUENCE
```

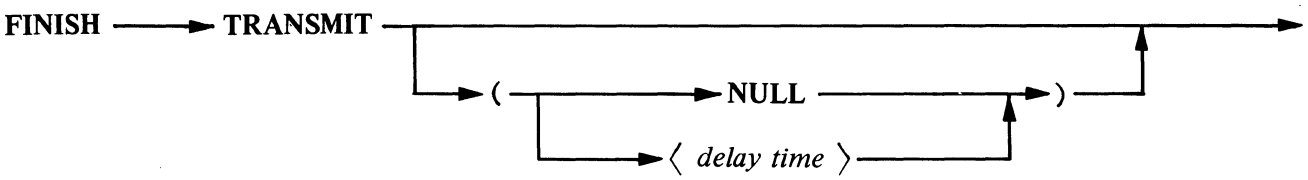
is equivalent to:

```
            INITIALIZE SEQUENCE.
GETSEQ:     FETCH SEQUENCE [ENDOFSEQ].
            TRANSMIT CHARACTER.
            GO TO GETSEQ.
ENDOFSEQ:
```

Prior to execution of this statement, the **DCP** sequence number logic must be initialized via the Initialize Sequence Statement.

FINISH STATEMENT

Syntax



Examples

```
FINISH TRANSMIT.
FINISH TRANSMIT (NULL).
FINISH TRANSMIT (3 SEC).
```

Semantics

The purpose of the ⟨*finish statement*⟩ is to take a line out of the transmit ready state and prepare the line to receive information. The adapter cluster delays a period of time long enough for the last character **TRANSMIT**ted to be transmitted, plus 2 milliseconds, before the line is put in a receive ready state. Although the ⟨*finish statement*⟩ puts the line in a receive ready state, the cluster hardware invokes a delay that inhibits any data from being received for 25 milliseconds. An **INITIATE RECEIVE** construct should precede any subsequent ⟨*receive statement*⟩, to override the 25 millisecond hardware delay.

The ⟨*delay time*⟩ option allows the programmer to specify a software delay of ⟨*time*⟩ before execution continues in the ⟨*control definition*⟩.

For example, the statement

```
FINISH TRANSMIT (3 SEC).
```

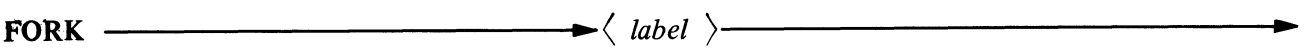
is equivalent to

```
FINISH TRANSMIT.
DELAY (3 SEC).
```

The **FINISH TRANSMIT (NULL)** construct is equivalent to **FINISH TRANSMIT.**

FORK STATEMENT

Syntax



Example

```
FORK 10.
```

Semantics

The ⟨*fork statement*⟩ is allowed in only those ⟨*control definition*⟩*s* and ⟨*request definition*⟩*s* that are written to communicate with full duplex terminal types. This statement can be executed in the ⟨*control definition*⟩ or ⟨*request definition*⟩ of the auxiliary line or the primary line. The execution of this statement causes the co-line control, if not busy, to branch to and begin executing code in the ⟨*control definition*⟩ that executes the **FORK** at the ⟨*label*⟩ specified, while control on the **FORK**ing line executes an implicit **PAUSE** (i.e., a ⟨*pause statement*⟩) and continues executing in parallel. The co-line is determined busy or not busy by testing the **BUSY** bit (i.e., **LINE(BUSY)** or **AUX(LINE(BUSY))**, whichever is appropriate). If the co-line is busy, the ⟨*fork statement*⟩ acts as a no-op.

Pragmatics

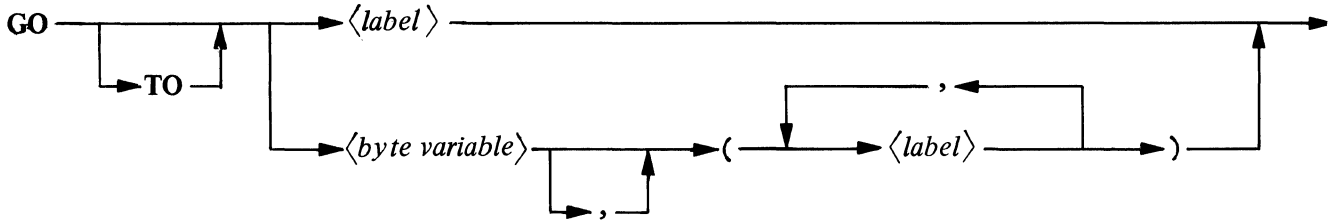
Synchronization problems can occur between the primary and auxiliary lines as a result of the ⟨*fork statement*⟩ executing the implicit **PAUSE**. The implicit **PAUSE** yields use of the DCP, to allow processing to proceed on other lines. Thus, processing on the co-line is actually started before the **FORK**ing line exits the ⟨*fork statement*⟩. As a result, the programmer must, by some means (e.g., by setting and testing line **TOG**s), effect the synchronization of the lines. This is especially critical if the code contains ⟨*wait statement*⟩*s* and ⟨*continue statement*⟩*s*. The following example illustrates how full duplex lines could "hang" as a result of poor synchronization.

```
        FORK 10.
        WAIT.
          .
          .
          .
10:     CONTINUE.
        WAIT.
          .
          .
          .
```

Assume that the primary line executes the **FORK 10.** At that point, the primary line temporarily yields use of the DCP to other lines. The auxiliary line starts up and executes the **CONTINUE.** Since primary control is still at the ⟨*fork statement*⟩ and is not in a ⟨*wait statement*⟩, the auxiliary line **CONTINUE** acts as a no-op. Next, the auxiliary line executes the **WAIT.** When the primary line is given use of the processor again, it executes its **WAIT.** At this point, the primary and auxiliary lines are "hung," each **WAIT**ing for a **CONTINUE** from its co-line.

GO TO STATEMENT

Syntax



Examples

```
GO GETNEXTCHAR.
GO 10.
GO TO 10.
GO TO TOGS, (0, 1, 2, 3).
GO TO STATION (5, 9, 12).
```

Semantics

The ⟨*go to statement*⟩ alters the path of control, that is, the sequential flow of statement execution, within a ⟨*control definition*⟩.

GO TO ⟨*label*⟩

This form of the ⟨*go to statement*⟩ unconditionally transfers control to the ⟨*label*⟩ specified.

GO TO ⟨*byte variable*⟩ . . .

This form of the ⟨*go to statement*⟩ provides a convenient means of dynamically selecting one or more ⟨*label*⟩*s* to which control could branch. The ⟨*label*⟩ to branch to is selected by using the ⟨*byte variable*⟩ as an index value. If N represents the number of ⟨*label*⟩*s* in the ⟨*go to statement*⟩, then the ⟨*label*⟩*s* are numbered 0 to N-1. The ⟨*label*⟩ corresponding to the index value is the ⟨*label*⟩ to which control branches. If the index value is greater than N-1, then control continues at the statement following the ⟨*go to statement*⟩.

Supplementary Example

```
GO TO STATION (5, 9, 12).
% EXECUTION CONTINUES HERE IF STATION > 2.
      .
      .
      .
5:    TOG [0] = TRUE.
      .
      .
      .
9:    TOG [1] = TRUE.
      .
      .
      .
12:   TOG [2] = TRUE.
      .
      .
```

Go To Statement – Continued

This example illustrates the "**GO TO** ⟨*byte variable*⟩ . . ." option of the ⟨*go to statement*⟩. The value of **STATION** determines the next statement to be executed. If the value of **STATION** is 0, control branches to the ⟨*label*⟩ 5; if the value of **STATION** is 1, control branches to ⟨*label*⟩ 9; and if the value of **STATION** is 2, control branches to ⟨*label*⟩ 12. If the value of **STATION** is greater than 2, control continues at the next sequential statement.

IDLE STATEMENT

Syntax

IDLE ——————————————————→

Semantics

The execution of the ⟨*idle statement*⟩ causes a logical line to be suspended in an idle state. Specifically, **IDLE** causes the **LINE(BUSY)** ⟨*bit variable*⟩ to be set **FALSE**, the line to be suspended in a "sleeping" and "ready" status, and all subsequent inbound data to be discarded.

Pragmatics

The ⟨*idle statement*⟩ suspends the execution of a ⟨*control definition*⟩ for a logical line. Normally, this statement should be executed only when there are no outbound messages queued for any stations on the line and none of the stations on the line are **ENABLED** for input (or possibly, if the programmer wants any available inbound data discarded). Consider the following example of the contention-type ⟨*control definition*⟩ taken from the Burroughs SYMBOL/SOURCENDL program.

```
CONTROL CONTENTION:

    INITIATE REQUEST.
    INITIATE ENABLEINPUT.
    IDLE.
```

In this example, **IDLE** is executed only after it has been determined (by means of **INITIATE REQUEST** and **INITIATE ENABLEINPUT**) that the station is not **QUEUED** and not **ENABLED** for input.

Once a line is in an idle state, the line remains in an idle state until one of the following circumstances occurs:

a. If the line **TYPE** is **DIALIN** and the line becomes connected (as a result of **ANSWER = TRUE** in the ⟨*line definition*⟩, a **DIALOUT (TYPE = 98) DCWRITE** from the MCS, or an **ANSWER THE PHONE (TYPE = 100) DCWRITE** from the MCS), the ⟨*control definition*⟩ is initiated for the line.

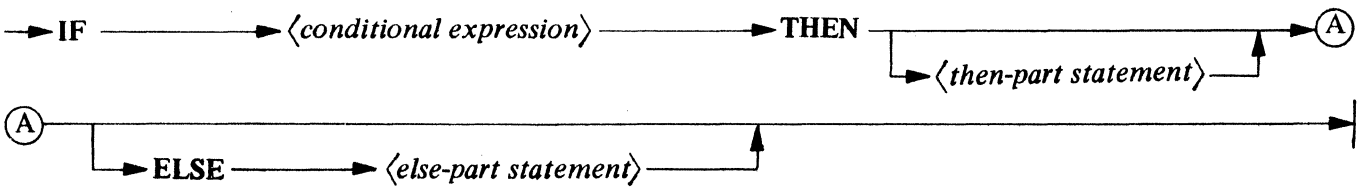
b. If any of the following station-oriented **DCWRITE**s are executed for any station assigned to the line, then the ⟨*control definition*⟩ is initiated for the line.

   1. **ENABLE INPUT (TYPE = 35)**
   2. **DISABLE INPUT (TYPE = 36)**
   3. **SET CHARACTERS (TYPE = 39)**
   4. **SET TRANSMISSION NUMBER (TYPE = 40)**
   5. **SET/RESET LOGICALACK (TYPE = 43)**
   6. **NULL STATION REQUEST (TYPE = 48)**
   7. **SET/RESET SEQUENCE MODE (TYPE = 49)**

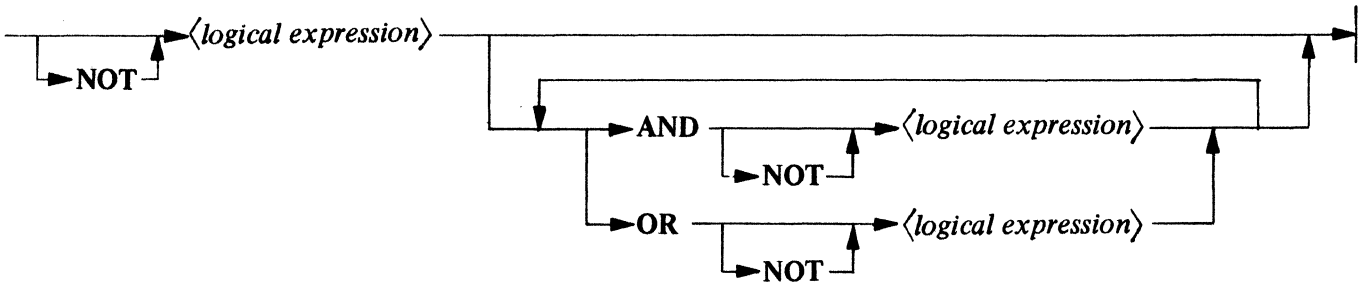
c. If a **WRITE** request or a **READ** request is found in the DCP's Request Queue (placed there as a result of the MCS executing a **WRITE (TYPE = 33) DCWRITE** or a **READ-ONCE ONLY (TYPE = 34) DCWRITE**, or the I/O intrinsics) for a station on the line, then the appropriate ⟨*request definition*⟩ is initiated for the line.

IF STATEMENT

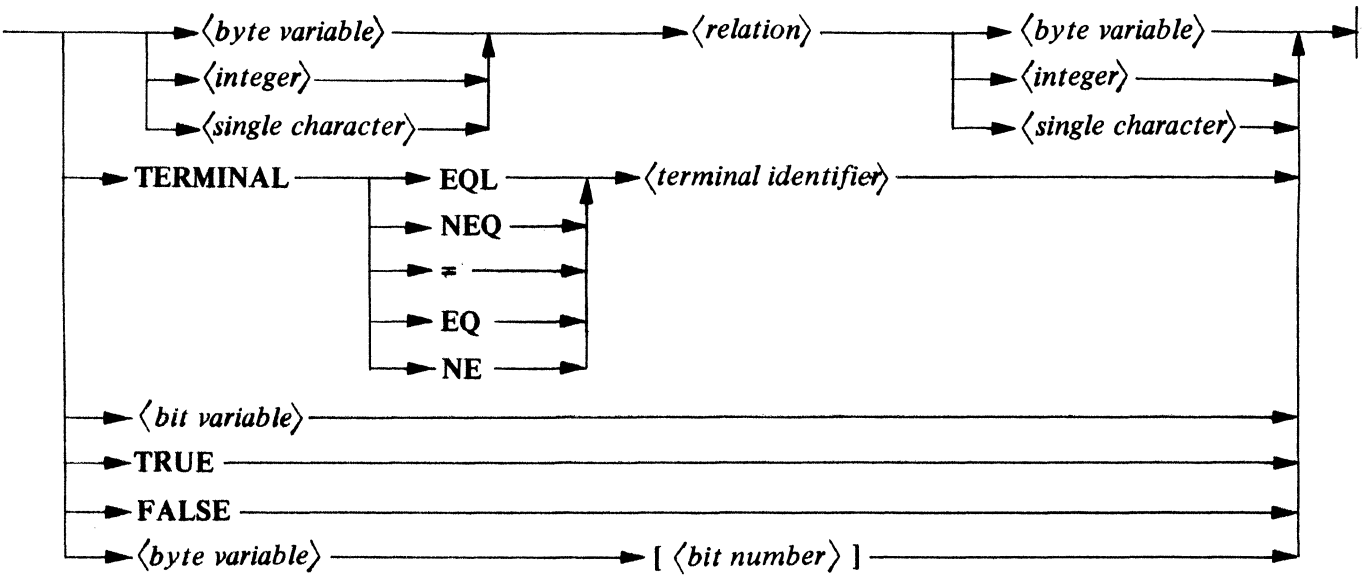
Syntax



⟨conditional expression⟩



⟨logical expression⟩



Examples

```
IF TRUE THEN.

IF TOG[2] AND TALLY[1] = 25 THEN GO TO 99.

IF TOG[0] THEN TOG[0] = FALSE.

IF TALLY[0] LSS TALLY[1] THEN TALLY[0] = TALLY[1].

IF TERMINAL = TELETYPE THEN DELAY (150 MILLI).
```

⟨*terminal identifier*⟩ has the syntactic form of ⟨*identifier*⟩ and is the name used in a subsequent **TERMINAL** definition which describes the physical attributes of a particular terminal type.

Note that no parentheses are allowed in the conditional expression, and that the normal precedence of "**NOT**" over "**AND**" over "**OR**" applies. The code generated for testing the condition is such that only those logical expressions needed to establish the truth or falsity of the condition are evaluated. The only time where evaluation of a logical expression (or the lack thereof) should matter is when **RECEIVE ADDRESS** is used as a byte variable.

```
IF CHARACTER = 4 "FF" THEN
    INITIATE BREAK

ELSE
    BEGIN
    CHARACTER = 4 "00".
    GO TO 0.
    END.

IF STATION (READY) THEN
    IF STATION (QUEUED) THEN
        LINE (TOG[0]) = TRUE
    ELSE
        GO TO 10
ELSE IDLE.
```

Semantics

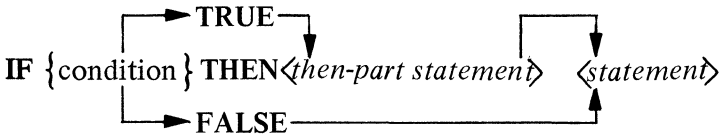
The ⟨*if statement*⟩ causes a condition (i.e., a Boolean expression) to be evaluated. The subsequent path of program control depends on whether the condition is evaluated as **TRUE** or **FALSE**.

If the condition is **TRUE**, the ⟨*then-part statement*⟩ following the **THEN**, if present, is executed. Program control then resumes at a statement that follows the ⟨*if statement*⟩.

If the condition is **FALSE**, the ⟨*else-part statement*⟩ following the **ELSE** is executed or, if the **ELSE** ⟨*else-part statement*⟩ is omitted, program control resumes at the ⟨*statement*⟩ following the ⟨*if statement*⟩.

If Statement – Continued

The ⟨*control statement*⟩ can be any legal ⟨*control statement*⟩, including the ⟨*if statement*⟩ and ⟨*compound statement*⟩. The meanings of the relational operators are contained in table 5-1. The following diagrams illustrate the ⟨*if statement*⟩ semantics.



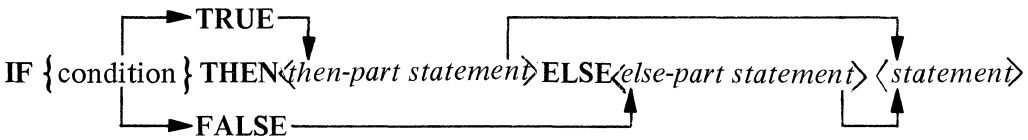
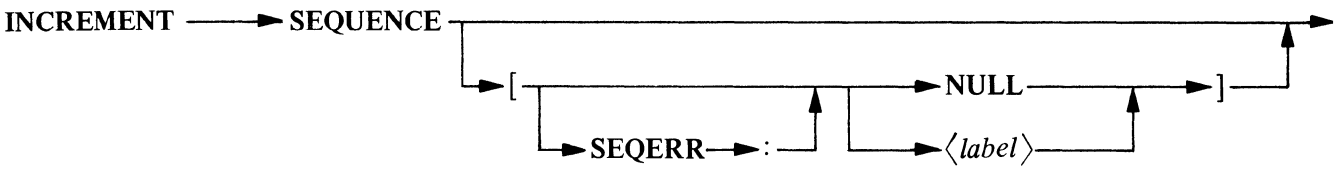


Table 5-1. Relational Operators

RELATIONAL OPERATOR	MEANING	SYNONYMS
LSS	Less than	<and LS
LEQ	Less than or equal to	LE
EQL	Equal to	= and EQ
NEQ	Not equal to	NE
GEQ	Greater than or equal to	GE
GTR	Greater than	>and GT

INCREMENT STATEMENT

Syntax



Examples

```
INCREMENT SEQUENCE.
INCREMENT SEQUENCE [SEQERR: NULL].
INCREMENT SEQUENCE [NULL].
INCREMENT SEQUENCE [999].
```

Semantics

The ⟨*increment statement*⟩ causes the sequence number stored in the DCP Station Table to be increased by the value of the increment (also stored in the DCP Station Table), providing that the station is in sequence mode; otherwise, this statement is a no-op.

When using the **INCREMENT SEQUENCE** construct, provision should be made for taking action if the increment caused the sequence number to exceed (overflow) the size of the sequence number field. The programmer can take such action by including the optional syntax. Failure to include overflow action results in an implicit **TERMINATE ERROR** if an overflow occurs.

The **SEQERR:NULL** and **NULL** options are semantically equivalent. These options set the **SEQERR** ⟨*bit variable*⟩ **TRUE**, and control continues at the next sequential instruction.

The **SEQERR:**⟨*label*⟩ and ⟨*label*⟩ options are semantically equivalent. They cause the **SEQERR** ⟨*bit variable*⟩ to be set **TRUE**, and control to branch to ⟨*label*⟩.

Regardless of whether error action is specified or not, an overflow of the sequence number field destroys the contents of that field.

Pragmatics

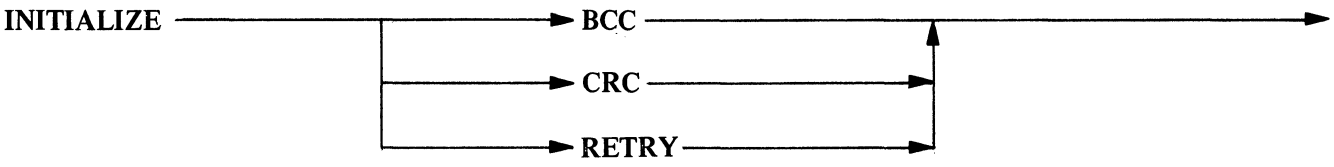
SEQUENCE MODE

A station is considered to be in sequence mode whenever its **SEQUENCE** ⟨*bit variable*⟩ toggle is **TRUE**. **SEQUENCE** can be set **TRUE** only as a result of the controlling MCS executing the **SET/RESET SEQUENCE MODE (TYPE = 49) DCWRITE.** In addition, the **TYPE 49 DCWRITE** also stores the starting sequence number and increment in the appropriate fields of the DCP Station Table.

Sequence mode can be used for any application that the NDL programmer may see fit. Its use, however, requires common conventions between the NDL programmer and the MCS programmer. Burroughs has utilized sequence mode constructs in two ⟨*request definition*⟩*s* of SYMBOL/SOURCENDL: READTELETYPE and WRITETELETYPE. Both require the cooperation of SYSTEM/CANDE to effect the execution of those statements. The reader is referred to those ⟨*request definition*⟩*s* as an example of a particular application that Burroughs has implemented. Other statements relative to sequence mode are the ⟨*transmit statement*⟩ **(TRANSMIT SEQUENCE** construct) and the ⟨*store statement*⟩ **(STORE SEQUENCE** construct).

INITIALIZE STATEMENT

Syntax



Examples

```
INITIALIZE BCC.
INITIALIZE CRC.
INITIALIZE RETRY.
```

Semantics

INITIALIZE BCC

The **INITIALIZE BCC** construct causes the ⟨*byte variable*⟩ **BCC** to be initialized for purposes of accumulating a Block Check Character. The value to which **BCC** is initialized is dependent upon the horizontal parity defined for the station's associated ⟨*terminal definition*⟩ (in the ⟨*terminal parity statement*⟩). If horizontal parity is defined as **HORIZONTAL:ODD**, then **BCC** is initialized to all ones (i.e., **4"FF"**). If defined as **HORIZONTAL:EVEN**, then **INITIALIZE BCC** initializes **BCC** to all zeroes (i.e., **4"00"**).

INITIALIZE CRC

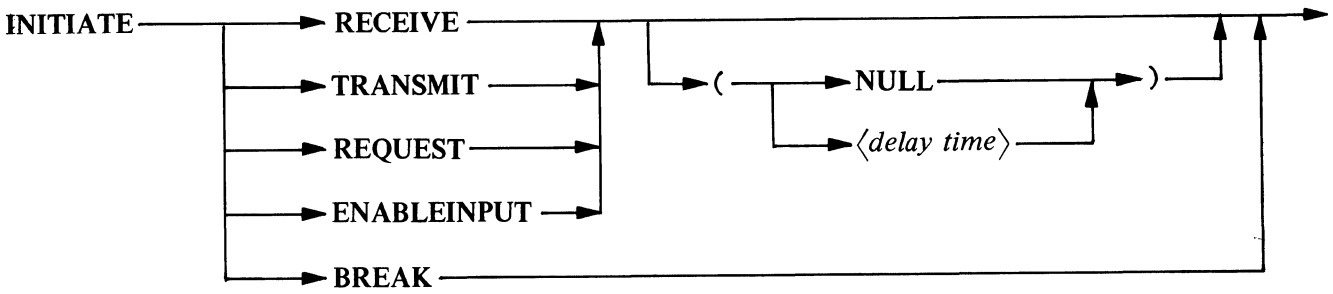
The **INITIALIZE CRC** construct initializes **CRC** to the initial value required for calculating the Cyclic Redundancy Check. Any ⟨*terminal definition*⟩ referencing a ⟨*control definition*⟩ (in the ⟨*terminal control statement*⟩) that contains this instruction must define the horizontal parity (in the ⟨*terminal parity statement*⟩) as **HORIZONTAL:CRC(16)**; otherwise a syntax error is generated.

INITIALIZE RETRY

The **INITIALIZE RETRY** construct causes the value stored in DCP INITIAL RETRY to be stored in DCP RETRY.

INITIATE STATEMENT

Syntax



Examples

```
INITIATE RECEIVE.
INITIATE TRANSMIT (3 SEC).
INITIATE REQUEST (NULL).
INITIATE ENABLEINPUT.
INITIATE BREAK.
```

Semantics

INITIATE RECEIVE

The **INITIATE RECEIVE** construct causes the adapter cluster to initiate a receive delay calculated for the station. After the delay, the hardware is ready to receive information.

The amount of time delayed, referred to as the Initiate Receive delay, is unique to each station and is calculated at compile-time for each station. The algorithm that the compiler uses to calculate the Initiate Receive delay is described in the following three paragraphs.

a. If the ⟨*modem definition* referenced in the ⟨*station definition*⟩ (in the ⟨*station modem statement*⟩ defines the modem **NOISEDELAY** as being greater than zero, then the Initiate Receive delay is 2 milliseconds less than the combined ⟨*time*⟩*s* defined in the ⟨*modem noisedelay statement*⟩ and the ⟨*modem transmitdelay statement*⟩.

b. If the modem **NOISEDELAY** is defined as zero and the modem **TRANSMITDELAY** is defined as being less than 7 milliseconds, then the Initiate Receive delay is zero.

c. If the modem **NOISEDELAY** is defined as zero and the modem **TRANSMITDELAY** is defined as being equal to or greater than 7 milliseconds, then the Initiate Receive delay is the lesser of

15 milliseconds or (1.5 milliseconds + $\frac{\text{modem } \textbf{TRANSMITDELAY}}{2}$).

The **NULL** option or the ⟨*delay time*⟩ option can be used to override the calculated Initiate Receive delay. **NULL** immediately readies the hardware so that it can receive information. ⟨*delay time*⟩ specifies a ⟨*time*⟩ to be used in place of the Initiate Receive delay.

The Initiate Receive delays for all stations assigned to a given line are maximized; the DCP stores this maximized value and uses it for the entire line.

Initiate Statement – Continued

Pragmatics

An **INITIATE RECEIVE** instruction should precede the first ⟨*receive statement*⟩ following a transmission. If it does not, there is a possibility that execution of the ⟨*receive statement*⟩ will be delayed for a period of time of up to 25 milliseconds. The cause of the 25-millisecond delay is described under the semantics of the ⟨*finish statement*⟩.

INITIATE TRANSMIT

The **INITIATE TRANSMIT** construct causes the Adapter Cluster to be put in a transmit state after a calculated delay. The amount of time delayed is referred to as the Initiate Transmit Delay, and is unique to each station. It is derived by taking the greater of the **NOISEDELAY** ⟨*time*⟩ specified for the modem configured at the system end, or the **TURNAROUND** ⟨*time*⟩ specified by the station's associated ⟨*terminal definition*⟩ . This construct must be executed prior to any attempt to transmit information.

The **NULL** option or the ⟨*delay time*⟩ option can be used to override the calculated Initiate Transmit delay. **NULL** causes the adapter cluster to be put in a transmit state immediately. ⟨*delay time*⟩ specifies a ⟨*time*⟩ to be used in place of the Initiate Transmit delay.

The Initiate Transmit delays for all stations assigned to a given time are maximized; the DCP stores this maximized value and uses it for the entire line.

INITIATE REQUEST

The **INITIATE REQUEST** construct conditionally initiates the next function as indicated by the message at the head of the Station Queue. The initiation of the function is conditional, subject to the following: the station must be valid, ready, and queued. Specifically, **STATION(VALID)**, **STATION(READY)**, and **STATION(QUEUED)** must be **TRUE**; otherwise, the instruction acts as a no-op.

The specific function invoked by this construct is dependent upon the type of message at the head of the Station Queue. Most commonly the message is a WRITE (TYPE=33) DCWRITE, thus causing the Transmit Request for the station to be entered. A READ-ONCE ONLY (TYPE=34) DCWRITE message at the head of the Station Queue would cause control to enter the Receive Request for the station. Other messages (unrelated to input or output) invoke their specific function and then transfer control to the beginning of the ⟨*control definition*⟩. For example, a SET SEQUENCE MODE (TYPE=49) DCWRITE message would cause control to enter the subroutine of the DCP that handles setting sequence mode and, when finished, control would be transferred to the beginning of the ⟨*control definition*⟩.

The ⟨*delay time*⟩ option allows the programmer to specify that an implicit ⟨*delay statement*⟩ for the ⟨*time*⟩ specified, be executed before initiation of the next function from the Station Queue. For example, the statement

```
INITIATE REQUEST (3 SEC).
```

is equivalent to

```
IF STATION(VALID) THEN

    IF STATION(READY) THEN

        IF STATION(QUEUED) THEN

            BEGIN
            DELAY(3 SEC).
            INITIATE REQUEST.
            END.
```

The **INITIATE REQUEST (NULL)** construct is equivalent to **INITIATE REQUEST.**

INITIATE ENABLEINPUT

The **INITIATE ENABLEINPUT** construct conditionally transfers control to the receive request appropriate for the station (that is, the station referenced by the ⟨*byte variable*⟩ named STATION). The transfer of control is conditional, subject to the following: the station must be valid, ready, and enabled for input. More specifically, **STATION(VALID)**, **STATION(READY)**, and **STATION(ENABLED)**, must be **TRUE**; otherwise, the instruction acts as a no-op.

The NDL programmer can initially enable a station for input by means of the ⟨*station enableinput statement*⟩. Additionally, after DCP initialization, the station's MCS can enable or disable the station for input by means of the **TYPES** 35 and 36 **DCWRITE**s.

(NULL) and (⟨*delay time*⟩) allow the programmer to specify that an implicit ⟨*delay statement*⟩, for *time* specified, be executed before the transfer of control. ⟨*delay time*⟩ has the syntactic form of ⟨*time*⟩. For example, the statement

```
INITIATE ENABLEINPUT (3 SEC).
```

is equivalent to:

```
IF STATION(VALID)  THEN
    IF STATION(READY)  THEN
        IF STATION(ENABLED)  THEN
            BEGIN
            DELAY (3 SEC).
            INITIATE ENABLEINPUT.
            END.
```

The **(NULL)** option specifies zero delay.

INITIATE BREAK

The **INITIATE BREAK** construct causes binary zeroes to be transmitted on the line, thus changing the state of the line to a "spacing" condition. The line remains in the spacing condition until some subsequent construct causes the adapter cluster to change the state of the line. Constructs that would change the line's state are **INITIATE TRANSMIT, INITIATE RECEIVE, FINISH TRANSMIT, BREAK,** and **IDLE.**

PAUSE STATEMENT

Syntax

PAUSE——————————————————→

Semantics

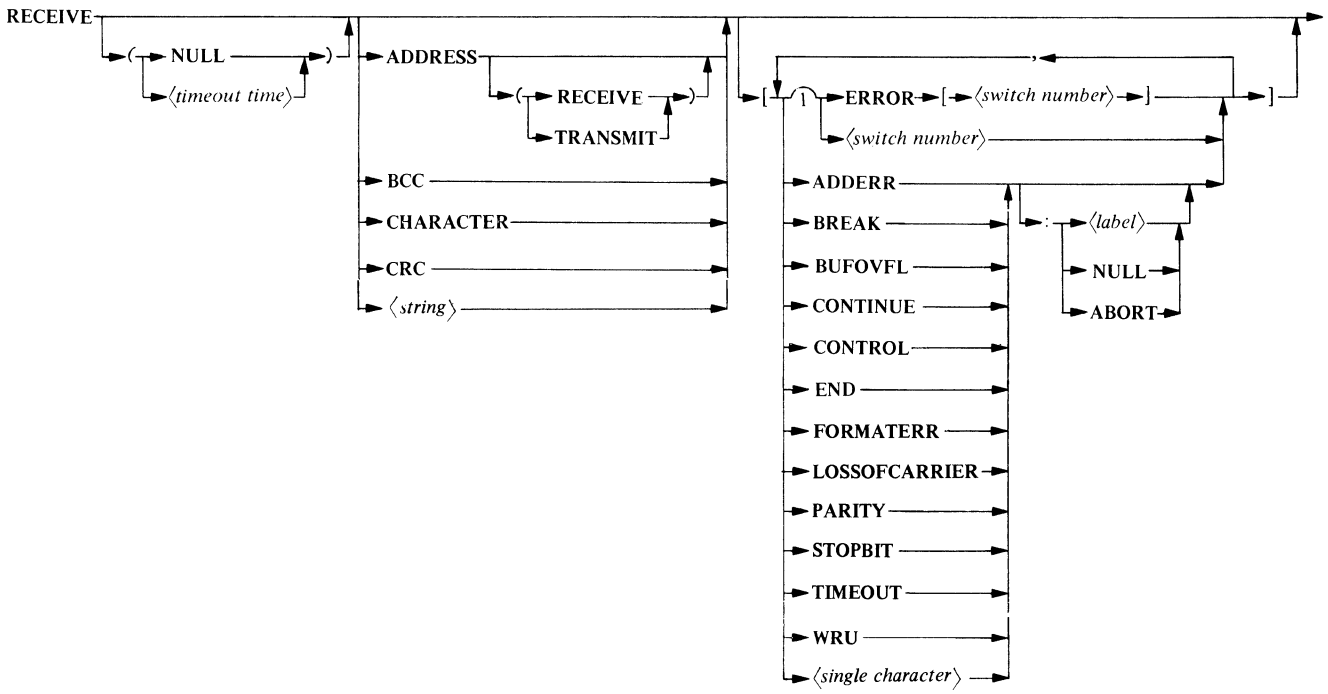
The ⟨*pause statement*⟩ suspends a ⟨*control definition*⟩ in a "sleep" state for a minimum period of time to allow the DCP to service other lines. It is recommended that a ⟨*pause statement*⟩ be used in any kind of loop that would tie up processor time and thereby prevent the servicing of other lines. The failure to do so results in a high number of timeout faults.

Pragmatics

Instances may occur in which the DCP requires an even greater period of "sleep" to service other lines. Repeated timeout faults, despite utilization of the ⟨*pause statement*⟩, are indications of such conditions. A greater period of "sleep" time can be effected by means of a ⟨*delay statement*⟩, with the ⟨*time*⟩ specified greater than "sleep" time effected by the ⟨*pause statement*⟩.

RECEIVE STATEMENT

Syntax



Examples

```
RECEIVE.

RECEIVE CHARACTER.

RECEIVE (3 SEC) ADDRESS (RECEIVE) [0, ADDERR:10].

RECEIVE (NULL) [
                    PARITY:999,
                    LOSS OF CARRIER:999,
                    END,
                    WRU:NULL
                    ].

RECEIVE CRC [ERROR [1], CRCERR:10].
```

Semantics

The ⟨*receive statement*⟩ causes the adapter cluster to attempt to receive information from the appropriate logical line.

The following two syntax items define a maximum amount of time that the adapter cluster should wait for receipt of the first character, and then each subsequent character, if applicable, before assuming that the terminal has "timed out." If neither of these options is included, the ⟨*timeout time*⟩ defined (in the ⟨*terminal timeout statement*⟩) for the station's ⟨*terminal definition*⟩ is implicitly used as the ⟨*timeout time*⟩ in this statement.

(NULL)

This option specifies that the adapter cluster should wait an infinite amount of time.

(⟨*timeout time*⟩**)**

The ⟨*timeout time*⟩ defines a ⟨*time*⟩ that the adapter cluster should wait for a character. If this ⟨*time*⟩ is exceeded before receipt of a character, and the **TIMEOUT** syntax appears, then the action specified for **TIMEOUT** is taken (refer to **TIMEOUT**). If the ⟨*timeout time*⟩ is exceeded and **TIMEOUT** syntax does not appear, an implicit **TERMINATE ERROR** is executed.

The following syntax options define the nature of the information to be received, the amount of information to be received, and how the information is to be handled. If these options are omitted, it is semantically equivalent to specifying **CHARACTER** (i.e., **"RECEIVE."** is semantically equivalent to **"RECEIVE CHARACTER."**)

ADDRESS

The proper number of address characters (as defined by the station's ⟨*terminal definition*⟩ in the ⟨*terminal address size statement*⟩) are received and checked for agreement against the actual address characters defined in the ⟨*station address statement*⟩. If the address characters do not correspond, an address error condition results . If the ADDERR syntax appears then the specified action is taken; otherwise, an implicit **TERMINATE ERROR** is executed. (Refer to the **ADDERR** semantics.)

ADDRESS (RECEIVE)

This option is equivalent to **ADDRESS**, except that **ADDRESS (RECEIVE)** must be used when an address pair is defined in the ⟨*station address statement*⟩ and the programmer needs to check for the proper receive address.

ADDRESS (TRANSMIT)

This option is equivalent to **ADDRESS**, except that **ADDRESS (TRANSMIT)** must be used when an address pair is defined in the ⟨*station address statement*⟩ and the programmer needs to check for the proper transmit address.

BCC

One character is received and checked against the ⟨*byte variable*⟩ **BCC**. If the character received and **BCC** are not equal, a Block Check Character error condition results. If the **BCCERR** syntax appears, then specified action is taken; otherwise an implicit **TERMINATE ERROR** is executed.

Presumably, if the **RECEIVE BCC** instruction appears, the programmer has defined horizontal parity in the ⟨*terminal parity statement*⟩, and the accumulated Block Check Character is contained in **BCC.**

CHARACTER

One character is received and stored in **CHARACTER.**

CRC

Two characters are received. The first character is checked against **CRC[0]**, and the second compared against **CRC[1]**. If the characters received and **CRC** are not equal, a Cyclic Redundancy Check error condition results. If the **CRCERR** syntax appears, then specified action is taken; otherwise an implicit **TERMINATE ERROR** is executed.

Presumably, if the **RECEIVE CRC** construct appears, the programmer has defined horizontal parity **HORIZONTAL:CRC(16)** in the ⟨*terminal parity statement*⟩, and the Cyclic Redundancy Check is contained in **CRC[0]** and **CRC[1]**.

⟨*string*⟩

The number of characters as indicated by the length of the ⟨*string*⟩ are received and checked against those characters in the ⟨*string*⟩. If the ⟨*string*⟩ and the characters received are not equal, then a format error condition results. If the **FORMATERR** syntax option appears, then that action is taken; otherwise an implicit **TERMINATE ERROR** is executed.

The following syntax options specify actions to be taken upon either the receipt of defined characters or occurrences of specific error conditions:

ERROR[⟨*switch number*⟩ **]**

Associates a previously defined Error Switch with the ⟨*receive statement*⟩. This allows the programmer to associate a set of previously defined error actions with the ⟨*receive statement*⟩, thus reducing the amount of coding required for each ⟨*receive statement*⟩. **BREAK, BUFOVFL, LOSSOFCARRIER, PARITY, STOPBIT,** and **TIMEOUT** syntax options are not allowed if the **ERROR[** ⟨*switch number*⟩ **]** syntax appears in the ⟨*receive statement*⟩. Refer to the ⟨*error switch statement*⟩ for more information.

⟨*switch number*⟩

Semantically equivalent to **ERROR[** ⟨*switch number*⟩ **]**.

ADDERR

The **ADDERR** option variations cause the following actions if an address error is detected when attempting to receive the address characters of a terminal:

ADDERR	sets **TRUE** the **ADDERR** ⟨*bit variable*⟩ and branches control to the next sequential statement.
ADDERR:NULL	sets **TRUE** the **ADDERR** ⟨*bit variable*⟩ . Execution proceeds as if the error condition did not occur.
ADDERR: ⟨*label*⟩	sets **TRUE** the **ADDERR** ⟨*bit variable*⟩ and branches control to ⟨*label*⟩.
ADDERR:ABORT	Not allowed.

BREAK

The **BREAK** option variations cause the following actions if a break, that is, at least two character-times of a spacing line condition, is detected by the adapter cluster while receiving:

BREAK	sets **TRUE** the ⟨*bit variable*⟩ **BREAK[RECEIVE]** , and branches control to the next sequential statement.
BREAK:NULL	sets **TRUE** the ⟨*bit variable*⟩ **BREAK[RECEIVE]** . Execution proceeds as if the break did not occur.
BREAK: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **BREAK [RECEIVE]** , and branches control to ⟨*label*⟩.
BREAK:ABORT	sets **TRUE** the ⟨*bit variable*⟩ **BREAK[RECEIVE]** , and executes an implicit **TERMINATE ERROR**.

BUFOVFL

The **BUFOVFL** option variations cause the following actions if the DCP is unable to service a Cluster Attention Needed (CAN) interrupt before the adapter cluster receives another character (thus destroying the previous character):

BUFOVFL	sets **TRUE** the ⟨*bit variable*⟩ **BUFOVFL**, and branches control to the next sequential statement.
BUFOVFL:NULL	sets **TRUE** the ⟨*bit variable*⟩ **BUFOVFL**. Execution proceeds as if the error condition did not occur.
BUFOVFL:⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **BUFOVFL**, and branches control to ⟨*label*⟩.
BUFOVFL:ABORT	sets **TRUE** the ⟨*bit variable*⟩ **BUFOVFL**, and executes an implicit **TERMINATE ERROR**.

CONTINUE

This option is allowed only in ⟨*receive statement*⟩*s* of ⟨*control definition*⟩*s* and ⟨*request definition*⟩*s* that are written to communicate with full duplex terminal types. **CONTINUE** syntax causes action as described below if the co-line executes a ⟨*continue statement*⟩ before all information specified by the ⟨*receive statement*⟩ is received.

CONTINUE	branches control to the next sequential statement.
CONTINUE:NULL	causes no action. Execution proceeds as if the ⟨*continue statement*⟩ had not been executed.
CONTINUE:⟨*label*⟩	branches control to ⟨*label*⟩.
CONTINUE:ABORT	Not allowed.

CONTROL

The **CONTROL** option variations cause the following actions if the control character of the station (as defined in the ⟨*station control character statement*⟩) is received:

CONTROL	sets **TRUE** the ⟨*bit variable*⟩ **CONTROLFLAG**, and branches control to the next sequential statement.
CONTROL:NULL	sets **TRUE** the ⟨*bit variable*⟩ **CONTROLFLAG**, and execution continues as if the character was not the station's control character.
CONTROL:⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **CONTROLFLAG**, and branches control to ⟨*label*⟩.
CONTROL:ABORT	Not allowed.

END

The **END** option variations cause the following actions if the "end" character of the station (as defined by the ⟨*terminal end character statement*⟩ in the ⟨*terminal definition*⟩ associated with the station) is received:

END	causes control to branch to the next sequential statement.
END:NULL	causes no action. Execution proceeds as if the character was not the "end" character.
END:⟨*label*⟩	branches control to ⟨*label*⟩.
END:ABORT	Not allowed.

FORMATERR

The following variations of the **FORMATERR** option cause the following actions if the characters received are not equal to those in the ⟨*string*⟩ (this item is appropriate only for the **RECEIVE** ⟨*string*⟩ construct of the ⟨*receive statement*⟩):

FORMATERR	sets **TRUE** the ⟨*bit variable*⟩ **FORMATERR**, and branches control to the next sequential statement.
FORMATERR:NULL	sets **TRUE** the ⟨*bit variable*⟩ **FORMATERR**. Execution proceeds as if the error did not occur.
FORMATERR: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **FORMATERR**, and branches control to ⟨*label*⟩ .
FORMATERR:ABORT	not allowed.

LOSSOFCARRIER

The **LOSSOFCARRIER** option variations cause the following actions if a loss of carrier is detected while receiving.

LOSSOFCARRIER	sets **TRUE** the ⟨*bit variable*⟩ **LOSSOFCARRIER**, and branches control to the next sequential statement.
LOSSOFCARRIER:NULL	sets **TRUE** the ⟨*bit variable*⟩ **LOSSOFCARRIER**. Execution proceeds as if the error did not occur.
LOSSOFCARRIER:⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **LOSSOFCARRIER**, and branches control to ⟨*label*⟩.
LOSSOFCARRIER:ABORT	sets **TRUE** the ⟨*bit variable*⟩ **LOSSOFCARRIER**, and executes an implicit **TERMINATE ERROR.**

There is one exception to the actions described above. If a loss of carrier is detected while receiving, and if the terminal is modem-connect, and if the terminal's ⟨*station definition*⟩ references a ⟨*modem definition*⟩ that contains the statement **LOSSOFCARRIER=DISCONNECT**, then an implicit disconnect is done, regardless of the action specified.

PARITY

The **PARITY** option variations cause the following actions if a parity bit error is detected by the adapter cluster:

PARITY	sets **TRUE** the ⟨*bit variable*⟩ **PARITY**, and branches control to the next sequential statement.
PARITY:NULL	sets **TRUE** the ⟨*bit variable*⟩ **PARITY**. Execution proceeds as if the error did not occur.
PARITY: *label*	sets **TRUE** the ⟨*bit variable*⟩ **PARITY**, and branches control to ⟨*label*⟩.
PARITY:ABORT	sets **TRUE** the ⟨*bit variable*⟩ **PARITY**, and executes a **TERMINATE ERROR.**

STOPBIT

The **STOPBIT** option variations cause the described actions if a stop bit error is detected by the adapter cluster:

STOPBIT	sets **TRUE** the ⟨*bit variable*⟩ **STOPBIT**, and branches control to the next sequential statement.
STOPBIT:NULL	sets **TRUE** the ⟨*bit variable*⟩ **STOPBIT**. Execution proceeds as if the error did not occur.
STOPBIT: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **STOPBIT**, and branches control to ⟨*label*⟩.
STOPBIT:ABORT	sets **TRUE** the ⟨*bit variable*⟩ **STOPBIT**, and executes a **TERMINATE ERROR.**

Receive Statement – Continued

TIMEOUT

The **TIMEOUT** option variations cause the actions described if the time required to receive a character exceeds the ⟨*timeout time*⟩. The ⟨*timeout time*⟩ is defined in the ⟨*terminal timeout statement*⟩, but can be overridden by including the (⟨*timeout time*⟩) or **(NULL)** syntax options in the ⟨*receive statement*⟩.

TIMEOUT	sets **TRUE** the ⟨*bit variable*⟩ **TIMEOUT**, and branches control to the next sequential statement.
TIMEOUT:NULL	sets **TRUE** the ⟨*bit variable*⟩ **TIMEOUT**. Execution proceeds as if the error did not occur.
TIMEOUT:⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **TIMEOUT**, and branches control to ⟨*label*⟩.
TIMEOUT:ABORT	sets **TRUE** the ⟨*bit variable*⟩ **TIMEOUT**, and executes a **TERMINATE ERROR**.

WRU

The **WRU** option causes the following actions if the **WRU** character of the station is received (the ⟨*station WRU character statement*⟩ defines the **WRU** character):

WRU	sets **TRUE** the **WRU** ⟨*bit variable*⟩, and branches control to the next sequential statement.
WRU:NULL	sets **TRUE** the **WRU** ⟨*bit variable*⟩, and execution proceeds as if the character received was not the WRU character.
WRU:⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ WRU, and branches control to ⟨*label*⟩.
WRU:ABORT	not allowed.

⟨*single character*⟩

The ⟨*single character*⟩ syntax causes the following actions if a character received is equal to the ⟨*single character*⟩ :

⟨*single character*⟩	branches control to the next sequential statement.
⟨*single character*⟩:NULL	causes no action. Execution proceeds as if the character received was not equal to the ⟨*single character*⟩.
⟨*single character*⟩:⟨*label*⟩	branches control to ⟨*label*⟩.
⟨*single character*⟩:**ABORT**	not allowed.

The allowable combinations of the ⟨*receive statement*⟩ syntax options are defined in table 5-2. The **(NULL)** and (⟨*timeout time*⟩) options are allowed in any form of the ⟨*receive statement*⟩. Allowed combinations of the other syntax options are denoted by a "X" in the appropriate columns and rows.

Supplementary Examples

Statement	Explanation
RECEIVE (3 SEC) [TIMEOUT:10].	Causes the adapter cluster to attempt to receive a character. If the character is not received within 3 seconds, the ⟨*bit variable*⟩ **TIMEOUT** is set TRUE and control branches to 10.
RECEIVE ADDRESS [ADDERR:99].	If the character(s) received do not equal those defined in the ⟨*station address statement*⟩, the ⟨*bit variable*⟩ **ADDERR** is set **TRUE**, and control branches to 99.
RECEIVE CHARACTER [CONTINUE:10, CONTROL:20, TIMEOUT:30, "*":40].	This statement would only be allowed in a ⟨*control definition*⟩ or ⟨*request definition*⟩ that is written to communicate with full duplex terminal types because it contains the **CONTINUE** option. **CONTINUE:10** would cause a branch to 10 if the co-line ⟨*control definition*⟩ executes a ⟨*continue statement*⟩ before a character is received. **CONTROL:20** would set **CONTROLFLAG TRUE** and branch to 20 if the character received is the station's control character. **TIMEOUT:30** would set **TIMEOUT TRUE** and branch to 30 if a character is not received within the ⟨*timeout time*⟩ defined in the ⟨*terminal timeout statement*⟩. **"*":40** would cause a branch to 40 if the character received is the asterisk character.

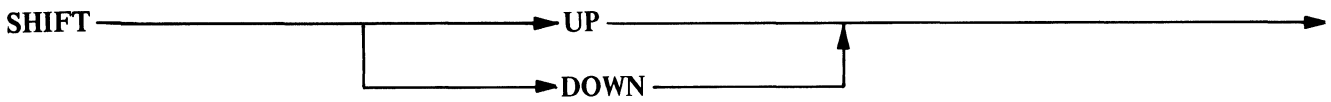
Statement	Explanation
RECEIVE[ERROR[0]].	An attempt is made to receive one character and store it in **CHARACTER**. If any errors described in the associated ⟨*error switch statement*⟩ occur while receiving, then the action defined in that ⟨*error switch statement*⟩ is taken.
RECEIVE[0]	Same as above.

Table 5–2. Allowable Combinations for ⟨*receive statement*⟩

	ADDERR	BCCERR	BREAK	BUFOVFL	CONTINUE	CONTROL	CRCERR	END	FORMATERR	LOSSOFCARRIER	PARITY	STOPBIT	TIMEOUT	TRANERR	WRU	⟨*single character*⟩
ADDRESS	X		X	X	X					X	X	X	X			
ADDRESS(RECEIVE)	X		X	X	X					X	X	X	X			
ADDRESS(TRANSMIT)	X		X	X	X					X	X	X	X			
BCC		X	X	X	X					X	X	X	X			
CHARACTER			X	X	X	X		X		X	X	X	X		X	X
CRC			X	X	X		X			X	X	X	X			
⟨*string*⟩			X	X	X				X	X	X	X	X			

SHIFT STATEMENT

Syntax



Semantics

The ⟨*shift statement*⟩ is to be used in a ⟨*control definition*⟩ that communicates with stations using the Baudot (5-bit) character code set. (The character code set is defined in the ⟨*terminal code statement*⟩ of the associated ⟨*terminal definition*⟩).

SHIFT UP indicates that received characters are to be translated to their respective uppercase graphics (usually referred to as FIGS).

SHIFT DOWN indicates that receive characters are to be translated to their respective lowercase graphics (usually referred to as LTRS).

If the station does not use Baudot code, the ⟨*shift statement*⟩ acts as a no-op.

Pragmatics

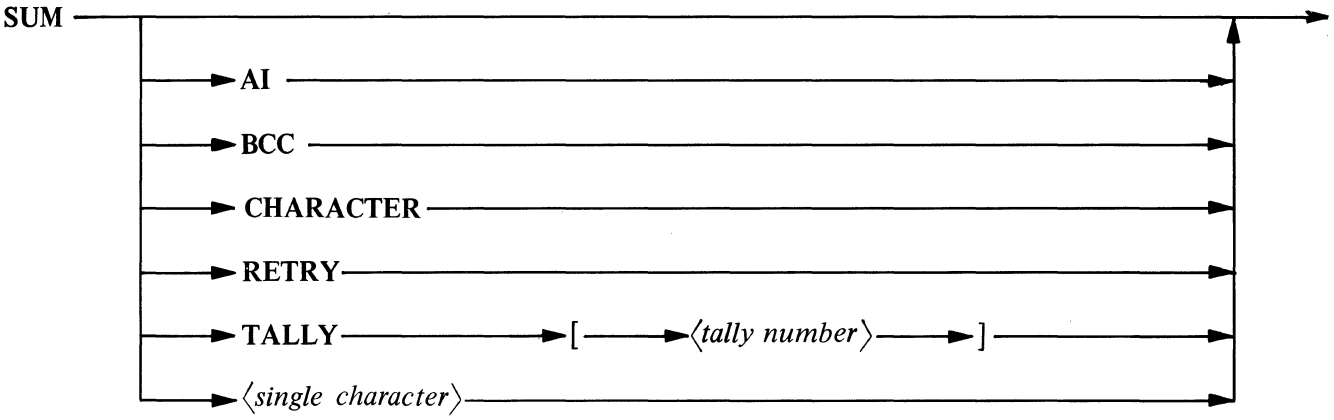
In the Baudot character code set, most bit patterns have two graphic representations: one is referred to as FIGS (the uppercase graphic), and the other as LTRS (the lowercase graphic).

When transmitting to a terminal that uses Baudot code, the terminal prints LTRS until it receives a specially designated character indicating that it should shift to printing FIGS. The terminal continues printing the FIGS until it receives a specially designated character indicating that it should resume printing the LTRS.

When the information is received from a terminal that uses Baudot, the same conventions hold true; that is, the terminal communicates whether FIGS or LTRS follow by the transmission of a designated character. The terminal initially transmits LTRS.

SUM STATEMENT

Syntax



Examples

```
SUM AI.
SUM CHARACTER.
SUM "A".
SUM TALLY [1].
```

Semantics

The purpose of the ⟨*sum statement*⟩ is to affect the calculation of the horizontal parity check (whether that be a Block Check Character or a Cyclic Redundancy Check). The specific effect of the ⟨*sum statement*⟩ is dependent upon two factors: the **SUM**med item, and whether the station's ⟨*terminal definition*⟩ for which ⟨*control definition*⟩ is running, defines horizontal parity as **CRC(16)**.

Following is a description of the effect that each form of the ⟨*sum statement*⟩ has on the calculation of the horizontal parity check. Any reference to **CRC** means **CRC[0]** and **CRC[1]** collectively.

SUM

Semantically equivalent to **SUM AI**.

SUM AI

If the horizontal parity check is a Block Check Character or is undefined, the contents of **AI** are exclusively OR-ed with the contents of **BCC**, and the result is stored in **BCC**.

If the horizontal parity check is a Cyclic Redundancy Check, the Cyclic Redundancy Check algorithm is computed with the contents of **AI** and **CRC**, and the result is stored in **CRC**.

SUM BCC

If the horizontal parity check is a Block Check Character or is undefined, then the contents of **BCC** are exclusively OR-ed with itself, and the result is stored in **BCC**. (The result in **BCC** would be zero in this case.)

If the horizontal parity check is a Cyclic Redundancy Check, the Cyclic Redundancy Check algorithm is computed with the contents of **CRC[0]** and **CRC**, and the result is stored in **CRC**.

SUM CHARACTER

If the horizontal parity check is a Block Check Character or is undefined, the contents of **CHARACTER** are exclusively OR-ed with the contents of **BCC**, and the result is stored in **BCC**.

If the horizontal parity check is a Cyclic Redundancy Check, the Cyclic Redundancy Check algorithm is computed with the contents of **CHARACTER** and **CRC**, and the result is stored in **CRC**.

SUM RETRY

If the horizontal parity check is a Block Check Character or is undefined, the contents of **RETRY** are exclusively OR-ed with the contents of **BCC**, and the result stored in **BCC**.

If the horizontal parity check is a Cyclic Redundancy Check, the Cyclic Redundancy Check algorithm is computed with the contents of **RETRY** and **CRC**, and the result is stored in **CRC**.

SUM TALLY [⟨*tally number*⟩]

If the horizontal parity check is a Block Check Character or is undefined, the contents of **TALLY** [⟨*tally number*⟩] are exclusively OR-ed with the contents of **BCC**, and the result is stored in **BCC**.

If the horizontal parity check is a Cyclic Redundancy Check, the Cyclic Redundancy Check algorithm is computed with the contents of **TALLY** [⟨*tally number*⟩] and the result is stored in **CRC**.

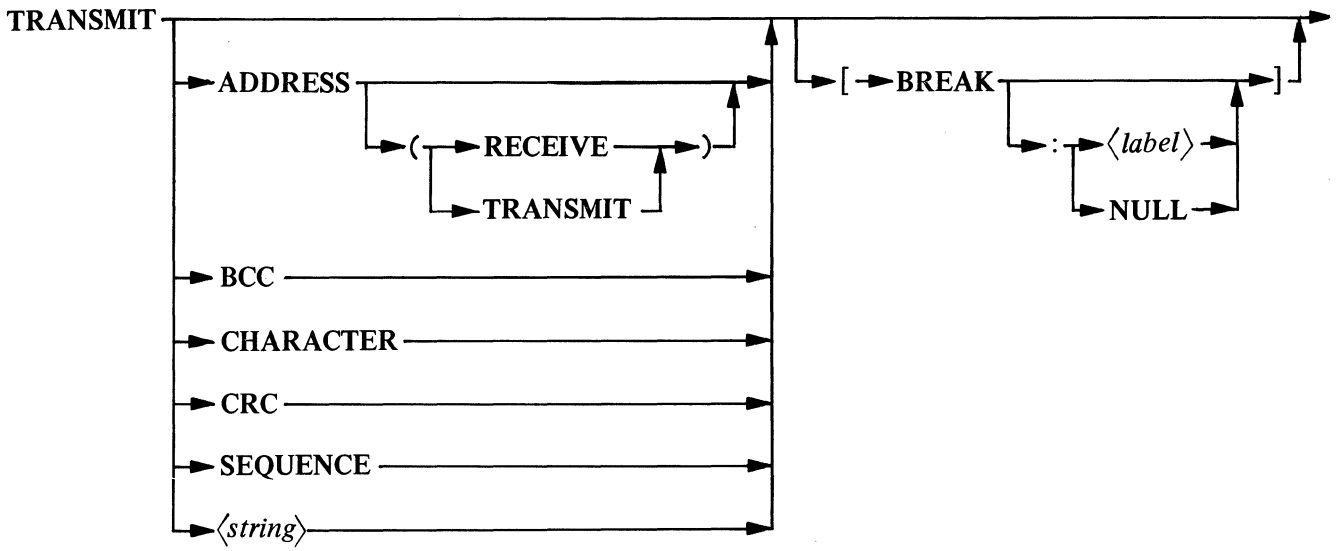
SUM ⟨*single character*⟩

If the horizontal parity check is a Block Check Character or is undefined, the ⟨*single character*⟩ is exclusively OR-ed with the contents of **BCC**, and the result is stored in **BCC**.

If the horizontal parity check is a Cyclic Redundancy Check, the Cyclic Redundancy Check, the Cyclic Redundancy Check algorithm is computed with the ⟨*single character*⟩ and **CRC**, and the result is stored in **CRC**.

TRANSMIT STATEMENT

Syntax



Examples

```
TRANSMIT.
TRANSMIT CHARACTER [BREAK:NULL].
TRANSMIT SOH STX 4"00"[BREAK:10].
TRANSMIT TRAN.
TRANSMIT ADDRESS (TRANSMIT) [BREAK].
```

Semantics

The ⟨*transmit statement*⟩ causes the adapter cluster to transmit information to a terminal. The following group of syntax options specifies the information to be transmitted. All options except **CHARACTER** use the ⟨*byte variable*⟩ **CHARACTER** as a temporary storage area; thus, any information contained in **CHARACTER** before execution of the ⟨*transmit statement*⟩ shall be destroyed by the ⟨*transmit statement*⟩. If none of the first group of options is chosen, it is semantically equivalent to specifying **CHARACTER** (i.e., "**TRANSMIT.**" is equivalent to "**TRANSMIT CHARACTER.**").

ADDRESS

The proper number of characters (as specified by the station's associated ⟨*terminal definition*⟩ in the ⟨*terminal address size statement*⟩) are taken from the address field in the Station Table and transmitted.

ADDRESS (RECEIVE)

This option is equivalent to **ADDRESS**, except that **ADDRESS (RECEIVE)** must be used when an address pair is defined in the ⟨*station address statement*⟩ and the programmer wants to transmit the receive address.

ADDRESS (TRANSMIT)

This option is equivalent to **ADDRESS**, except that **ADDRESS (TRANSMIT)** must be used when an address pair is defined in the ⟨*station address statement*⟩ and the programmer wants the transmit address transmitted.

BCC

The **BCC** option causes the contents of the ⟨*byte variable*⟩ **BCC** to be transmitted.

CHARACTER

The **CHARACTER** causes the contents of the ⟨*byte variable*⟩ **CHARACTER** to be transmitted.

CRC

This option causes two bytes to be transmitted; the contents of **CRC [0]** are transmitted first, followed by **CRC [1]**. If the station's associated ⟨*terminal definition*⟩ does not define horizontal parity as **CRC ([16])**, the use of this ⟨*option*⟩ causes a syntax error to be generated at compile time.

SEQUENCE

The **SEQUENCE** option causes the character representation of the value stored in the Sequence field of the Station Table to be transmitted if the station is in sequence mode (i.e., this ⟨*bit variable*⟩ **SEQUENCE** is **TRUE**); otherwise, the ⟨*transmit statement*⟩ is a no-op.

⟨*string*⟩

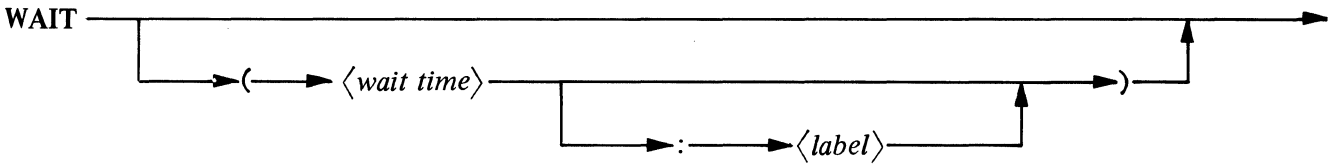
Each character of the ⟨*string*⟩ is transmitted.

The **BREAK** syntax allows the programmer to specify action if a "break" is received from the terminal while the adapter cluster is still transmitting. If this option is omitted and a break occurs, an implicit **TERMINATE ERROR** is executed. The following describes the actions of the three syntactical forms:

BREAK	sets **TRUE** the ⟨*bit variable*⟩ **BREAK [TRANSMIT]**, and causes a branch of control to the next statement.
BREAK: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **BREAK [TRANSMIT]**, and causes a branch of control to ⟨*label*⟩.
BREAK:NULL	sets **TRUE** the ⟨*bit variable*⟩ **BREAK [TRANSMIT]**. Execution proceeds as if the break did not occur.

WAIT STATEMENT

Syntax



Examples

```
WAIT.
WAIT (3 SEC).
WAIT (5 MILLI:6).
```

Semantics

The ⟨*wait statement*⟩ is only allowed in ⟨*control definition*⟩*s* that are written to communicate with full duplex terminal types. Execution of this statement causes the ⟨*control definition*⟩ to be suspended until the co-line executes a ⟨*continue statement*⟩. The optional syntax effects the statement as described below:

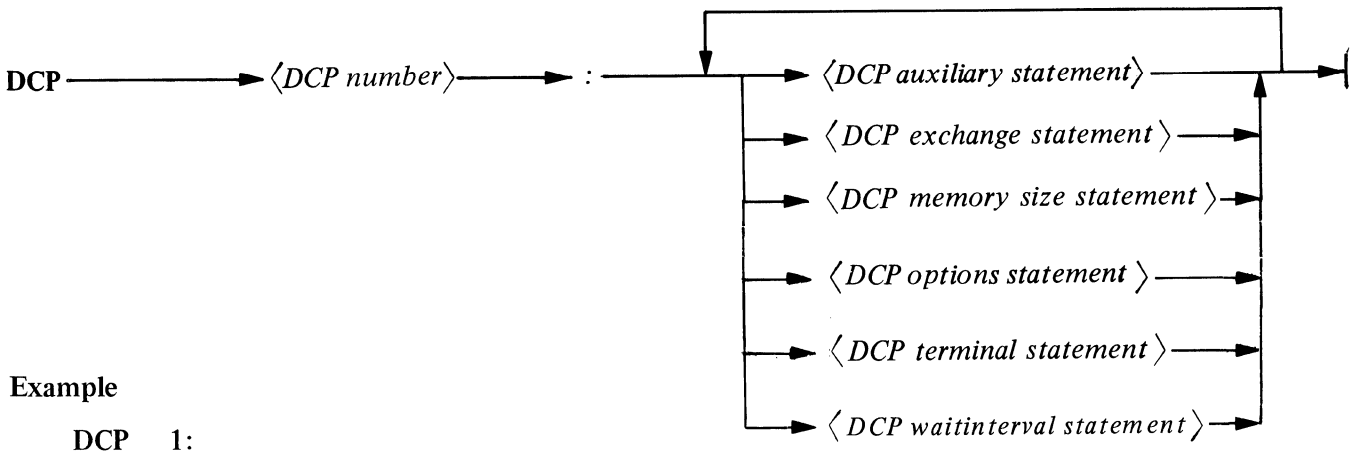
⟨*wait time*⟩	defines the maximum amount of ⟨*time*⟩ that the ⟨*control definition*⟩ should be suspending waiting for the ⟨*continue statement*⟩. If ⟨*wait time*⟩ is exceeded and the co-line has not executed a ⟨*continue statement*⟩, execution resumes at the next sequential statement.
⟨*wait time*⟩:⟨*label*⟩	same as above except execution resumes at ⟨*label*⟩ if a ⟨*continue statement*⟩ is not executed within ⟨*wait time*⟩.

Pragmatics

Refer to the ⟨*fork statement*⟩ pragmatics.

DCP DEFINITION

Syntax



Example

```
DCP   1:
      MEMORY     = 8196.
      EXCHANGE   = 2.
      TERMINAL   = SOMETERMINALNAME.
      OPTIONS    = LOCALTABLES.
```

Semantics

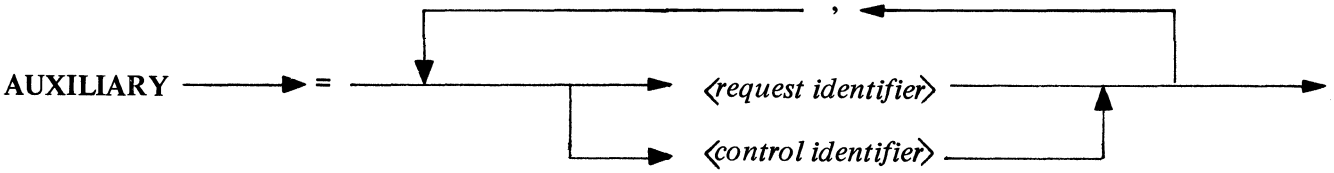
The ⟨*DCP definition*⟩ is the means by which the programmer defines attributes of each Data Communications Processor (DCP) in the Data Communications System.

The ⟨*DCP number*⟩ identifies the DCP and must correspond to an address (ranging from 0 through 7) wired into each DCP by the field engineer. The attributes of the DCP are defined subsequently by means of ⟨*DCP statement*⟩*s*. A maximum of six DCP definitions may appear in the NDL source program.

Each ⟨*DCP statement*⟩ is described subsequently.

AUXILIARY

Syntax



Semantics

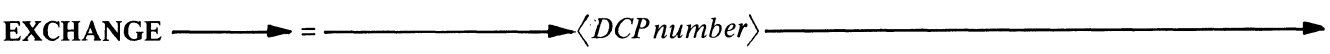
The auxiliary statement causes the specified request/control definitions to be stored in main memory. (The use of this capability is not recommended for B 7000 systems).

Example

```
DCP  0:  MEMORY    = 4096.
         OPTIONS   = LOCALTABLES.
         AUXILIARY = CONTENTION, PUNCHPAPERTAPE, ANALYZERASY.
DCP  1:  MEMORY    = 4096.
         AUXILIARY = ANALYZERASY, ANALYZERSYNC.
```

DCP EXCHANGE STATEMENT

Syntax



Example

```
EXCHANGE = 4.
```

Semantics

The ⟨*DCP exchange statement*⟩ specifies that the DCP shares hardware-exchanged adapter clusters with another DCP. ⟨*DCP number*⟩ defines the other DCP.

This statement is required in any ⟨*DCP definition*⟩ referenced by a ⟨*DCP exchange statement*⟩ in another ⟨*DCP definition*⟩, or in any ⟨*DCP definition*⟩ that does not have lines defined for it in the ⟨*line definition*⟩ section of the source program.

Pragmatics

The maximum number of DCPs that can share a set of adapter clusters is 2. The definitions of both DCPs that share adapter clusters must contain a ⟨*DCP exchange statement*⟩ naming the ⟨*DCP number*⟩ of the DCP with which it shares the adapter clusters. For example, if DCP 1 and DCP 2 share adapter clusters, then the definition of DCP 1 must contain the statement

```
EXCHANGE = 2.
```

and the definition of DCP 2 must contain the statement

```
EXCHANGE = 1.
```

If a DCP shares adapter clusters with another DCP, then any adapter cluster connected to either of the DCPs must be shared by both. A DCP is not allowed to share only a portion of its adapter clusters.

LINE SECTION REQUIREMENTS

If two DCPs share adapter clusters, it is required that the ⟨*line definition*⟩*s* for each DCP be given addresses (by means of a ⟨*line address statement*⟩) such that both DCPs do not have lines defined on the same cluster.

The following program segment would cause the compiler to generate a syntax error because both DCPs have lines defined on adapter cluster 0.

DCP TERMINAL REQUIREMENTS

If two DCPs are exchanged, their **TERMINAL** statements must be exactly the same.

DCP Exchange Statement – Continued

```
LINE L100:
    ADDRESS = 1:0:0. % ADDRESS = ⟨DCP⟩:⟨ADAPTER CLUSTER⟩:⟨LINE⟩.
        .
        .
        .
LINE L201:
    ADDRESS = 2:0:1.
        .
        .
        .
DCP 1:
    MEMORY   = 8192.
    EXCHANGE = 2.
DCP 2:
    MEMORY   = 8192.
    EXCHANGE = 1.
```

MCS RECONFIGURATION

The EXCHANGE CLUSTERS (TYPE = 129) DCWRITE function allows a Message Control System to transfer control of any or all adapter clusters, that are exchanged by two DCPs, from the DCP that currently controls the designated adapter clusters to the DCP with which it is exchanged. This aspect of the reconfiguration feature may be invoked in order to provide an installation with the ability to effect "load-leveling" between two DCPs that share hardware-exchanged adapter clusters or to transfer all of the work load of a DCP to its partner if the DCP malfunctions. For more information regarding reconfiguration, refer to the B 5000/B 6000/B 7000 Series DCALGOL Reference Manual.

Supplementary Example

The following is a program segment describing the data communications system illustrated in figure 5-1. This example illustrates how the ⟨*line definition*⟩ and ⟨*DCP definition*⟩ sections can be written to describe a data communications system in which two DCPs share hardware-exchanged adapter clusters.

```
%
%STATION DEFINITION SECTION.
%
     STATION DEFAULT ALLSTATIONS:

          .
          .
          .

     STATION STA1:

          DEFAULT  = ALLSTATIONS.
          TERMINAL = TTY.

     STATION STA2:

          DEFAULT  = ALLSTATIONS.
          TERMINAL = TTY.

     STATION STA3:

          DEFAULT  = ALLSTATIONS.
          TERMINAL = TTY.

     STATION STA4:

          DEFAULT  = ALLSTATIONS.
          TERMINAL = TTY.

     STATION STA5:

          DEFAULT  = ALLSTATIONS.
          TERMINAL = TTY.

     STATION STA6:

          DEFAULT  = ALLSTATIONS.
          TERMINAL = TTY.

%
%LINE DEFINITION SECTION.
%
%%%%%%%%%%%%%%    LINES FOR DCP 0    %%%%%%%%%%%%%%%%%%%%%%%%
                         LINE L000:

                              ADDRESS  = 0:0:0.
                              ADAPTER  = 1(DIRECT).
                              STATION  = STA1.

                         LINE L001:

                              ADDRESS  = 0:0:1.
                              ADAPTER  = 1(DIRECT).
                              STATION  = STA2.
```

```
                    LINE L020:

                        ADDRESS  = 0:2:0.
                        ADAPTER  = 1(DIRECT).
                        STATION  = STA5.

                    LINE L021:

                        ADDRESS  = 0:2:1.
                        ADAPTER  = 1(DIRECT).
                        STATION  = STA6.

%%%%%%%%%%%%%%%%%   LINES FOR DCP 1   %%%%%%%%%%%%%%%%%%%%

                    LINE L110:

                        ADDRESS  = 1:1:0.
                        ADAPTER  = 1(DIRECT).
                        STATION  = STA3.

                    LINE L111:

                        ADDRESS  = 1:1:1.
                        ADAPTER  = 1(DIRECT).
                        STATION  = STA4.

%
%DCP DEFINITION SECTION.
%

                    DCP 0:

                        MEMORY   = 8192.
                        EXCHANGE = 1.

                    DCP 1:

                        MEMORY   = 8192.
                        EXCHANGE = 0.
```

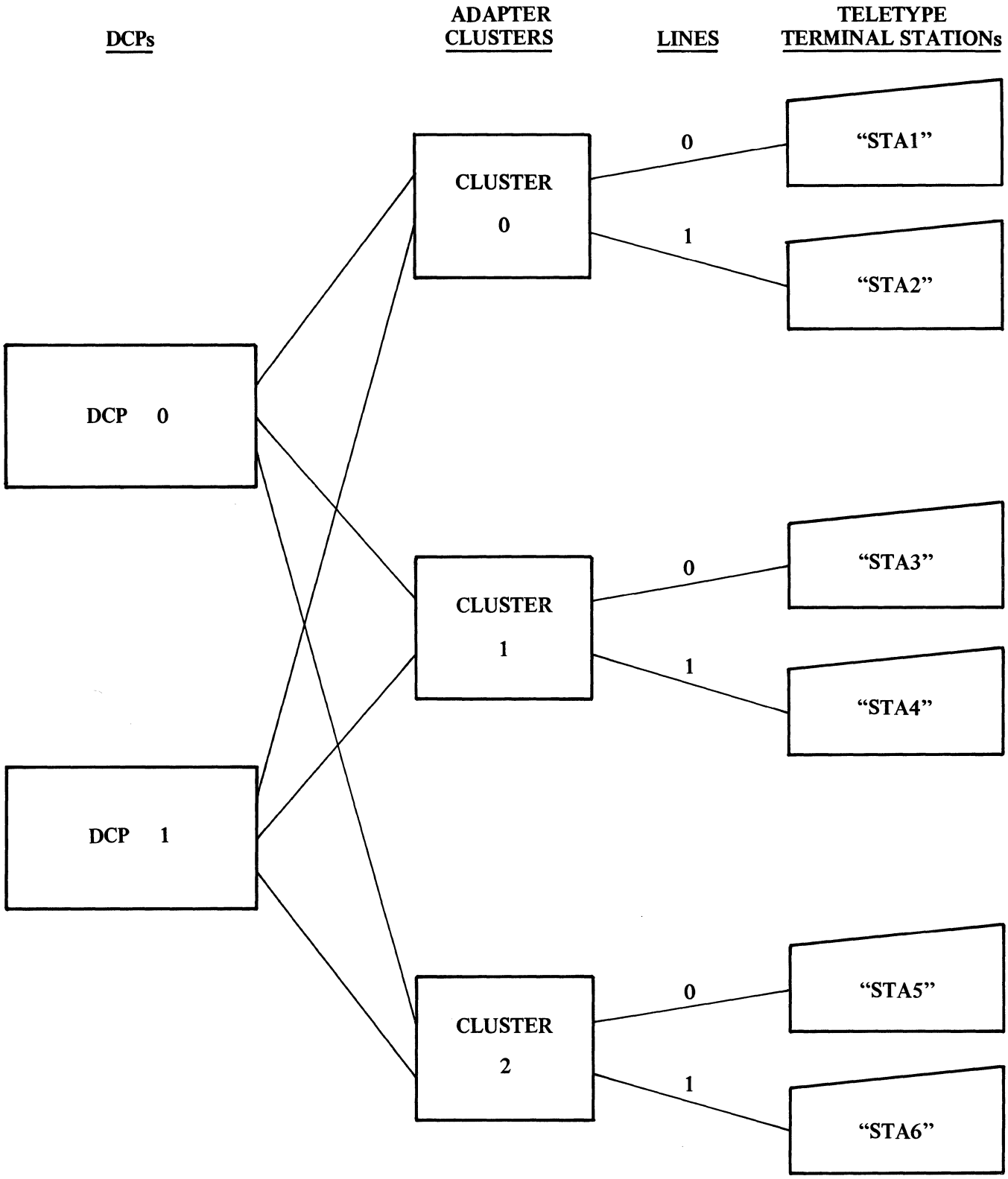
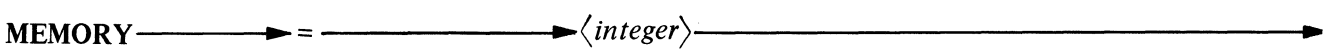


Figure 5-1. Adapter Clusters Exchange

DCP MEMORY SIZE STATEMENT

Syntax



Examples

```
MEMORY = 4096.
MEMORY = 0.
```

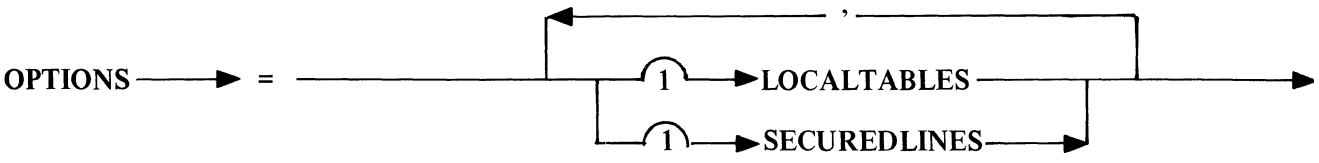
Semantics

The ⟨*DCP memory size statement*⟩ defines the number of words of local memory in the DCP being defined.

A zero value for ⟨*integer*⟩ indicates that the DCP has no local memory and that all code generated for the DCP shall reside in main system memory. A non-zero value for ⟨*integer*⟩ that is less than the amount of local memory required, as determined by the compiler, results in a compile-time error.

DCP OPTIONS STATEMENT

Syntax



Examples

```
OPTIONS = LOCALTABLES,
          SECUREDLINES.
```

Semantics

The **LOCALTABLES** option causes the **DCP** tables to be resident in **DCP** local memory rather than main system memory.

A **DCP** which has local memory tables may be declared exchanged with another **DCP** only if the second **DCP** also uses local memory tables.

The amount of memory space needed for **DCP** tables can be computed as follows:

Let **MAXLINE** = the maximum physical line address on the **DCP**
NLINES = the number of valid lines on the **DCP**
NSTAS = the sum of the station capacities for all lines on the **DCP**
MAXLSN = the largest **LSN** defined in the **NDL** source program

The total table space is:

(MAXLINE + 1) + (NLINES * 2) + (NSTAS) + (MAXLSN + 1) * 8

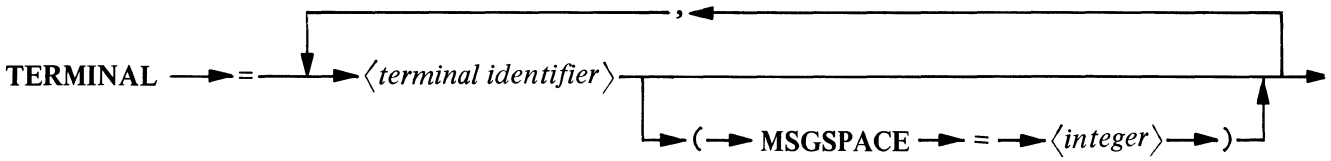
The **SECUREDLINES** option enables the **DCP** to detect the disconnection of a switched line under the following two circumstances:

(1) While the main system is performing a non-fatal memory dump with **DCP** tables in main memory.

(2) While a **DELAY** statement is being executed.

DCP TERMINAL STATEMENT

Syntax



Examples

```
TERMINAL = TELETYPE.
TERMINAL = M33, TD800 (MSGSPACE = 5), TELETYPE (MSGSPACE = 2).
```

Semantics

The purpose of the ⟨*DCP terminal statement*⟩ is twofold. Each aspect of this statement is discussed in the subsequent two paragraphs.

The primary purpose of the ⟨*DCP terminal statement*⟩ is to provide the means of specifying which terminal types in the data communications network that the DCP must be able to control. Only those terminal types specified in this statement will have the object code required to control them included in the object code generated for the DCP. If this statement is omitted from a DCP definition, the compiler includes the object code required to control all terminal types in the data communications network.

The second purpose of the ⟨*DCP terminal statement*⟩ is to provide a means of specifying the initial number of message spaces allotted for each terminal type controlled by the DCP.

The ⟨*terminal identifier*⟩ must name a terminal type defined by a ⟨*terminal definition*⟩, and specifies a terminal type for which the DCP must have access to the controlling code.

The **(MSGSPACE =** ⟨*integer*⟩**)** option specifies the number of message spaces initially allotted for the terminal type. If this option is omitted, two message spaces are allotted by default.

Pragmatics

Note that if any terminal type is not named in the ⟨*DCP terminal statement*⟩, the data communications network may not be reconfigured (by means of a reconfiguration DCWRITE in an MCS) such that it adds that terminal type to those terminal types controlled by the DCP. Refer to the supplementary example that follows.

Supplementary Example

The program segment below illustrates the pragmatics. A station whose terminal type is SCREENDEVICE cannot be added on the spare line L003 of DCP 1, because DCP 1 does not have the code available to control SCREENDEVICE.

```
%
%CONTROL & REQUEST DEFINITION SECTION.
%
        .
        .
        .
    REQUEST READTTY:
        .
        .
        .
    REQUEST WRITETTY:
        .
        .
        .
    REQUEST READSCREENDEVICE:
        .
        .
        .
    REQUEST WRITESCREENDEVICE:
        .
        .
        .
%
%TERMINAL DEFINITION SECTION.
%
    TERMINAL DEFAULT DEFAULTLIST:
        BLOCK          =   FALSE.
        SCREEN         =   FALSE.
        TURNAROUND     =   0.
        ICTDELAY       =   0.
        TRANSMISSION   =   0.
        DUPLEX         =   FALSE.
        TIMEOUT        =   3 SEC.
        ADDRESS        =   0.
        PAGE           =   0.
        CODE           =   ASC67.
        INHIBITSYNC    =   FALSE.
        BUFFER         =   NULL.
        MAXINPUT       =   80.
        WIDTH          =   72.
        PARITY         =   NULL.
        ADAPTER        =   4.
        WRU            =   ENQ.
        END            =   ETX(DYNAMIC).
        BACKSPACE      =   BS(DYNAMIC).
        CONTROL        =   CONTENTION.
```

{ The object code generated from these statements is required to control **TTY** terminal types. }

{ The object code generated from these statements is required to control **SCREENDEVICE** terminal types. }

```
TERMINAL TTY:
    DEFAULT      =  DEFAULTLIST.
    REQUEST      =  WRITETTY:TRANSMIT,READTTY:RECEIVE.

TERMINAL SCREENDEVICE:
    DEFAULT      =  DEFAULTLIST.
    SCREEN       =  TRUE.
    REQUEST      =  WRITESCREENDEVICE:TRANSMIT,READSCREENDEVICE:
                    RECEIVE.
```

These statements specify the ⟨*request definitions*⟩ required to control the defined terminal type. The object code generated by the procedures named here must be accessible by a **DCP** that has the terminal type attached to any of its lines.

```
%
%STATION DEFINITION SECTION.
%
    STATION DEFAULT ALLSTATIONS:
        ENABLEINPUT  =  TRUE.
        LOGICALACK   =  FALSE.
        MCS          =  SYSTEM/CANDE.
        CONTROL      =  QM.
        RETRY        =  15.
        MYUSE        =  OUTPUT,INPUT.
    STATION STA1:
        DEFAULT      =  ALLSTATIONS.
        TERMINAL     =  TTY.
    STATION STA2:
        DEFAULT      =  ALLSTATIONS.
        TERMINAL     =  TTY.
    STATION STA3:
        DEFAULT      =  ALLSTATIONS.
        TERMINAL     =  TTY.
    STATION STA4:
        DEFAULT      =  ALLSTATIONS.
        TERMINAL     =  SCREENDEVICE.
    STATION STA5:
        DEFAULT      =  ALLSTATIONS.
        TERMINAL     =  SCREENDEVICE.
    STATION STA6:
        DEFAULT      =  ALLSTATIONS.
        TERMINAL     =  SCREENDEVICE.
```

A ⟨*line definition*⟩ naming any of these stations must be a ⟨*line definition*⟩ for **DCP 0**. **DCP 1** does not have access to code required to control terminals associated with these stations.

A ⟨*line definition*⟩ naming any of these stations must be a ⟨*line definition*⟩ for **DCP 1**. **DCP 0** does not have access to code required to control terminals associated with these stations.

```
%
%LINE DEFINITION SECTION.
%%%%%%%%%%%%%%    LINES FOR DCP 0    %%%%%%%%%%%%%%%%%%%%%%%%

    LINE L000:

        ADDRESS       =   0:0:0.
        ADAPTER       =   1(DIRECT).
        STATION       =   STA1.

    LINE L001:

        ADDRESS       =   0:0:1.
        ADAPTER       =   1(DIRECT).
        STATION       =   STA2.

    LINE L002:

        ADDRESS       =   0:0:2.
        ADAPTER       =   1(DIRECT).
        STATION       =   STA3.

    LINE L003:  % THIS IS A SPARE LINE

        ADDRESS       =   0:0:3.
        MAXSTATIONS   =   1.

%%%%%%%%%%%%%%%%    LINES FOR DCP 1    %%%%%%%%%%%%%%%%%%%%%%%%

    LINE L100:

        ADDRESS       =   1:0:0.
        ADAPTER       =   1(DIRECT)
        STATION       =   STA4.

    LINE L101:

        ADDRESS       =   1:0:1.
        ADAPTER       =   1(DIRECT).
        STATION       =   STA5.

    LINE L102:

        ADDRESS       =   1:0:2.
        ADAPTER       =   1(DIRECT).
        STATION       =   STA6.

%
%DCP DEFINITION SECTION.
%

    DCP 0:

        MEMORY        =   8192.
        TERMINAL      =   TTY.

    DCP 1:

        MEMORY        =   8192.
        TERMINAL      =   SCREENDEVICE.
```

The ⟨*line station statement*⟩ of any ⟨*line definition*⟩ for DCP 0 must name a station that has a TTY terminal type associated with it. DCP 0 does not have access to code required to control SCREENDEVICE terminal types.

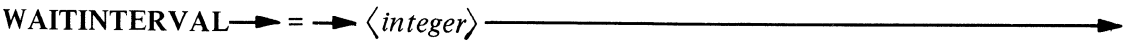
The ⟨*line station statement*⟩ of any ⟨*line definition*⟩ for DCP 1 must name a station that has a SCREENDEVICE terminal type associated with it. DCP 1 does not have access to code required to control TTY terminal types.

The effect of this statement is that this **DCP** has access to control code for **TTY** terminal types only.

The effect of this statement is that this **DCP** has access to control code for **SCREENDEVICE** terminal types only.

DCP WAITINTERVAL STATEMENT

Syntax



Example

```
WAITINTERVAL = 100.
```

Semantics

The **WAITINTERVAL** statement allows the site to specify the maximum time interval to wait between inserting a message into an empty result queue and interrupting the host machine. ⟨*Integer*⟩ is in milliseconds. A value of 0, or not using the **WAITINTERVAL**, gives the (old) behavior of causing an interrupt immediately after insertion into an empty result queue.

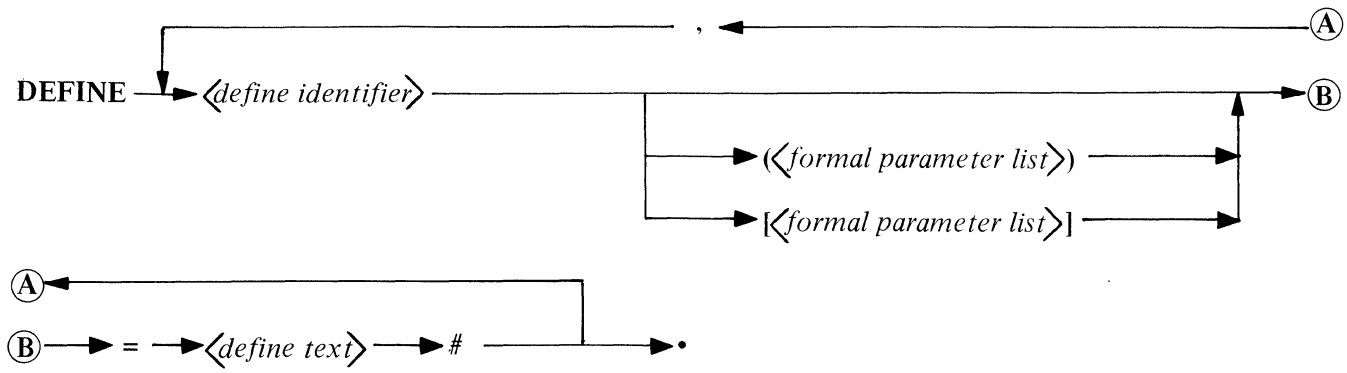
⟨*Integer*⟩ must be less than 4,000 milliseconds (4 seconds) and is truncated to a multiple of 1/60 of a second, for example, (⟨*integer*⟩) DIV (1000/60) * (1000/60). The **LOCALTABLES** options must be set. If the DCP does not have a time-of-day clock, the **WAITINTERVAL** setting is ignored.

A longer wait interval decreases the communication overhead on the main frame but increases the response time. The input and output messages per second through a DCP must exceed 1000 messages per waitinterval for any decrease in the overhead to occur; the larger the difference, the greater the decrease. A wait interval of 100 milliseconds should not noticeably increase the response time but limits the number of DCP interrupts to 10 per second.

Normally, when the DCP is idle, it waits for a message from the host system or for a cluster to need attention. If a non-zero wait interval is specified, the DCP will loop seeing if there is a system message, if any clusters need attention, or if the time interval has expired. This looping will increase slightly the time the DCP takes to respond to a system message or cluster attentions.

DEFINE DEFINITION

Syntax



Examples

```
DEFINE  CHARCOUNT = LINE(TALLY[0])#,
        NEEDACK   = 2#.

DEFINE  DELETE BLANKS (BYTE CTR, TEMP BYTE, BREAK LABEL) =

    BEGIN
    IF CHARACTER   = "    " THEN
        BEGIN
        BYTECTR    = BYTECTR +1.
        PAUSE.
        END
    ELSE
        BEGIN
        TEMPBYTE   = CHARACTER.

DELETELOOP:
        IF BYTECTR      = 0 THEN
            BEGIN
            CHARACTER = TEMPBYTE.
            TRANSMIT CHARACTER [BREAK:BREAKLABEL].
            END
        ELSE
            BEGIN
            BYTECTR    = BYTECTR-1.
            TRANSMIT "    " [BREAK:BREAKLABEL].
            GO TO DELETELOOP.
            END.
            END.
        END # .
```

Semantics

The ⟨*define definition*⟩ equates one or more ⟨*identifier*⟩s with ⟨*text*⟩s. Any subsequent appearance of the ⟨*define identifier*⟩ is syntactically and semantically equivalent to the ⟨*define text*⟩.

The ⟨*define definition*⟩ has two forms of syntax, the "simple" define and the "parametric" define. The parametric define has a series of parameters enclosed in parentheses or brackets. The simple define does not.

The appearance of a parametric define causes the actual parameters to replace the formal parameters in the associated ⟨*define text*⟩. Then, as with the simple define, the text logically replaces the ⟨*define identifier*⟩.

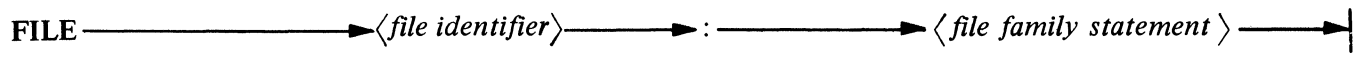
The ⟨*define text*⟩ is bracketed on the left by an equal sign (=) and on the right by a pound sign (#). The ⟨*text*⟩ can be any sequence of characters not containing a "free" pound sign. A free pound sign is any pound sign not in a ⟨*string*⟩ or ⟨*remark*⟩ . Unlike ALGOL, NDL ⟨*define definitions*⟩s may not be nested in any way.

⟨*define definitions*⟩ appearing before any **CONTROL** or **REQUEST** definitions are global to these and any subsequent definitions. ⟨*define definitions*⟩ appearing within a **REQUEST** or **CONTROL** definition are local to the definition and may redefine global ⟨*define definitions*⟩.

⟨*define definitions*⟩s may reference other ⟨*define definitions*⟩s in any mutually non-recursive way.

FILE DEFINITION

Syntax



Example

```
FILE NETWORK:  FAMILY = STATIONID1, STATION ID2, FILEID1.
```

Semantics

The ⟨*file definition*⟩ provides the means to define a data communications file and specify the stations associated with that file. The ⟨*file identifier*⟩ is the external name (TITLE) of the file, and has the syntactical form of a ⟨*system identifier*⟩.

A single-station file is a file that has one station associated with it. A single-station file can, but need not, be formally defined in a ⟨*file definition*⟩. The reason that a single-station file does not need to be defined is that each station is itself a file. The external name (TITLE) of such a file would be the ⟨*station identifier*⟩ of the station.

A multi-station file is, as the name implies, a file that has more than one station associated with it. Multi-station files must be defined in ⟨*file definition*⟩*s*.

Pragmatics

A general discussion of data communication files and their peculiarities can be found in chapter 2 of the B 7000/B 6000 Input/Output Subsystem Reference Manual under the heading "DATA COMM FILES." The information contained in that discussion is a prerequisite to understanding the significance of ⟨*file definition*⟩s. Chapter 3 of the same manual contains a table that lists all file attributes and provides an explanation of each attribute. Attributes relative to data communication files are found by examining the "KIND" column of the table for the key word "Datacom." The information found in the explanation of each data communications-relative attribute is also a prerequisite.

A detailed discussion of data communications object job I/O can be found in Section 6 of the B 5000/B 6000/B 7000 Series DCALGOL Reference Manual under the semantics of STATION ASSIGNMENT TO FILE (TYPE = 64). The information found there is not considered a prerequisite; however, it does contribute toward a deeper understanding of data communications files and data communications object job I/O.

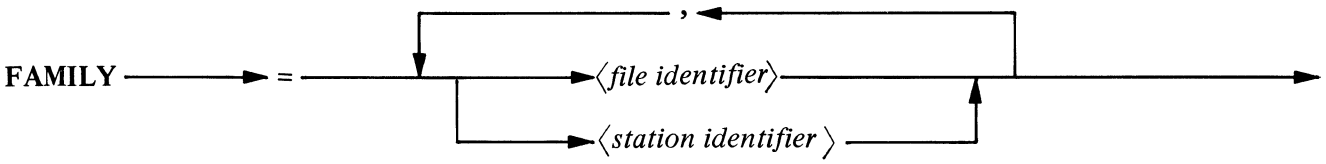
Definitions

FILE

File Family Statement

FILE FAMILY STATEMENT

Syntax



Example

```
FAMILY = STATIONID1, STATIONID2, FILEID1.
```

Semantics

The ⟨*file family statement*⟩ defines the stations associated with a data communications file. If a ⟨*file identifier*⟩ is named, all of the stations associated with the file named will also be associated with the file being defined. Any duplication of an ⟨*identifier*⟩ in a ⟨*file family statement*⟩ is ignored.

Supplementary Example

The following example is the ⟨*file definition*⟩ section of a hypothetical NDL program. Assume that the stations STATION1, STATION2, STATION3, STATION4, STATION5, STATION6, STATION7, and STATION8 have been defined in the ⟨*station definition*⟩ section.

```
FILE TTYS:
    FAMILY = STATION1, STATION2, STATION3.
```

TTYS is the ⟨*identifier*⟩ of this file. The **FAMILYSIZE** is 3. **STATION1**, **STATION2**, and **STATION3** are the stations associated with this file.

```
FILE CRTS:
    FAMILY = STATION4, STATION5, STATION6.
```

CRTS also has a **FAMILYSIZE** of 3. The stations associated with the file are **STATION4**, **STATION5**, and **STATION6**.

```
FILE EXECUTIVES:
  FAMILY = STATION1, CRTS, STATION6, STATION 7.
```

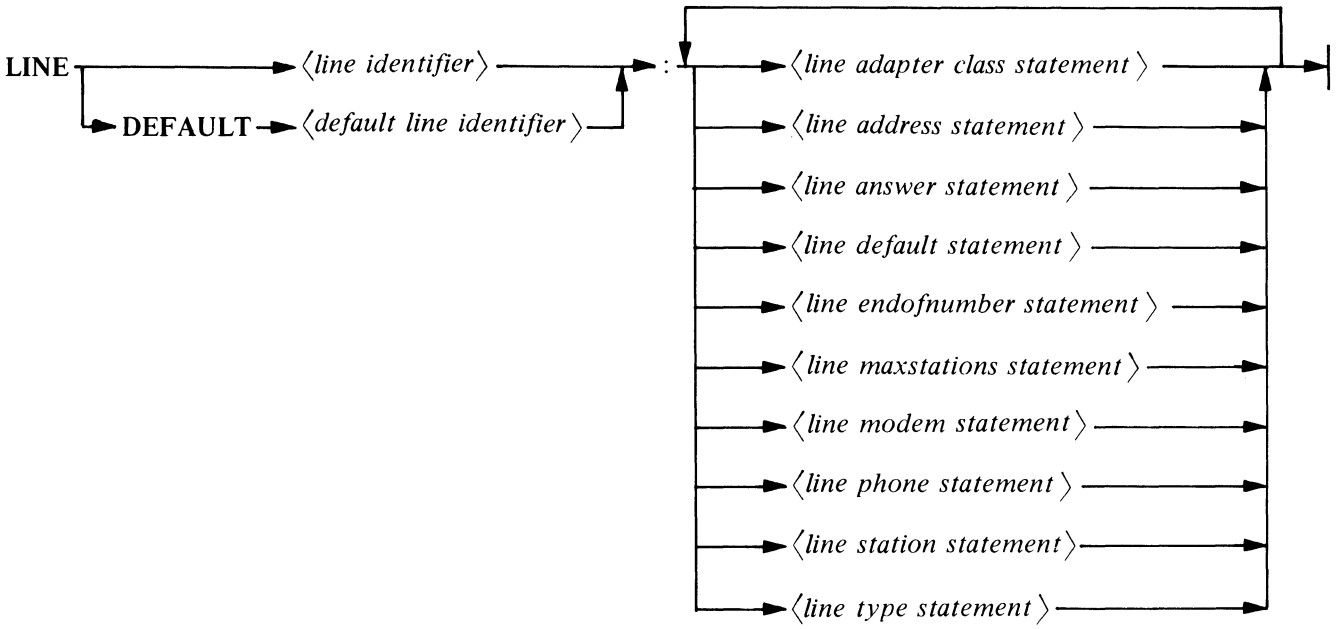
EXECUTIVES has a **FAMILYSIZE** of 5. The stations associated with the file are **STATION1**, **STATION4**, **STATION5**, **STATION6**, and **STATION7**.

```
FILE THE/ENTIRE/NETWORK:
    FAMILY = STATION7, STATION8, TTYS, CRTS.
```

THE/ENTIRE/NETWORK has a **FAMILYSIZE** of 8. The stations associated with **THE/ENTIRE/NETWORK** are **STATION1**, **STATION2**, **STATION3**, **STATION4**, **STATION5**, **STATION6**, **STATION7**, and **STATION8**.

LINE DEFINITION

Syntax



Examples

```
LINE TTYDIALIN:
    TYPE            = DIALIN.
    ADAPTER         = 1 (MODEM).
    MODEM           = TTYMODEM.
    ANSWER          = TRUE.
    PHONE           = 2139686521.
    ADDRESS         = 0:0:0.
    STATION         = TTYSTATION.
    MAXSTATIONS     = 1.

LINE DEFAULT LINEDEFAULTLIST1:
    ADAPTER         = 1 (MODEM).
    ANSWER          = TRUE.
    ENDOFNUMBER     = FALSE.
    MAXSTATIONS     = 1.
    TYPE            = DIALIN.
    MODEM           = TTYMODEM.
```

Semantics

⟨*line identifier*⟩ and ⟨*default line identifier*⟩ both have the same syntactical form as ⟨*identifier*⟩.

Each form of the ⟨*line definition*⟩ syntax is described subsequently.

LINE ⟨*line identifier*⟩ : . . .

This form of the ⟨*line definition*⟩ defines the attributes of a logical line in the data communications network. Line attributes are defined in one of the following ways:

a. Each attribute is defined explicitly by means of a ⟨*line statement*⟩ in the ⟨*line definition*⟩.
b. Each attribute is defined implicitly by an explicit reference to a set of default attribute values.
c. Some of the line attributes are defined implicitly as in b, and the remainder are defined explictly as in a.

Some ⟨*line statement*⟩*s* must be defined for each ⟨*line definition*⟩; others do not. Some of the statements may or may not require definition, depending upon the appearance of other statements. The semantics portion of each ⟨*line statement*⟩ states, among other things, whether the attribute must be defined and its effect upon the requirements of other attribute definitions.

To define the attributes of a line as described in item a above, this syntax form must be used.

To define the attributes of a line as described in items b and c above, this syntax form, the following syntax form, and the ⟨*line default statement*⟩ must be used in conjunction (this is described under the following syntax form).

LINE DEFAULT ⟨*default line identifier*⟩ : . . .

This syntax form is referred to as a Default ⟨*line definition*⟩. Its purpose is to decrease the number of source statements required to define all of the logical lines in the data communications system. This is accomplished in the following manner. Attributes common to several logical lines are defined by means of a Default ⟨*line definition*⟩. Associated with each Default ⟨*line definition*⟩ is a ⟨*default line identifier*⟩. Subsequent to the Default ⟨*line definition*⟩, any ⟨*line definition*⟩ that has those attributes in common can reference the ⟨*default line identifier*⟩, instead of repeating the list. (A ⟨*default line identifier*⟩ is referenced by means of a ⟨*line default statement*⟩.) The NDL compiler uses the last definition of a line attribute, and therefore the programmer can reference a Default ⟨*line definition*⟩ and change any attributes by redefining them in the ⟨*line definition*⟩.

In appearance, the Default ⟨*line definition*⟩ is similar to the ⟨*line definition*⟩. The differences are that the reserved word DEFAULT follows the reserved word LINE, and that there are no statements that are required to be defined in a Default ⟨*line definition*⟩.

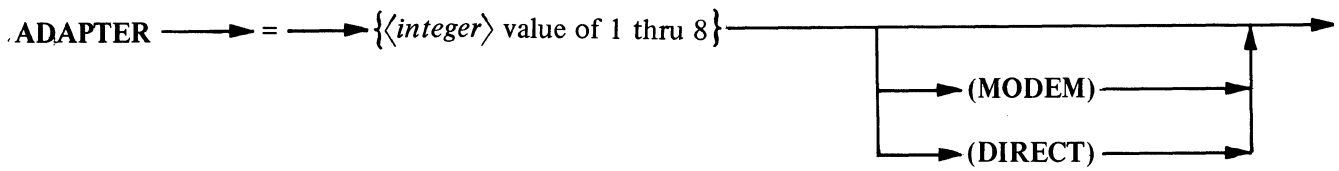
The following statements cannot be part of a default ⟨*line definition*⟩:

⟨*line address statement*⟩
⟨*line answer statement*⟩
⟨*line commandblock statement*⟩
⟨*line endofnumber statement*⟩
⟨*line station statement*⟩
⟨*line type statement*⟩

The above line statements are defined in this manual except the line commandblock statement which is defined in the B 6000/B 7000 Message - Oriented Data Comm. (MODC) Info. Manual (5011836).

LINE ADAPTER CLASS STATEMENT

Syntax



Examples

ADAPTER = 5.

ADAPTER = 4 (MODEM).

Semantics

The ⟨*line adapter class statement*⟩ identifies the Adapter Class of the line adapter for the logical line and, optionally, names the connection type (i.e., modem connect or direct connect).

The Adapter Class must be compatible with the ⟨*communication type number*⟩ specified in the ⟨*station adapter statement*⟩ of any station assigned to the line. (Note that all stations assigned to a line must have the same ⟨*communication type number*⟩ defined.) Table 5-3 lists the compatible Adapter Classes for each ⟨*communication type number*⟩. For example, a line having stations assigned to it that define a ⟨*communication type number*⟩ of 4 can name as an Adapter Class either 1, 2, 3, 4, or 5 (refer to table 5-3). On the other hand, a line having a station assigned to it that defines 15 as the ⟨*communication type number*⟩ can name only 5 as an Adapter Class.

If the connection type is named in the statement, it is considered by the compiler as documentation only. The compiler determines whether the line adapter or a modem-connect line adapter, by the presence or absence of a ⟨*line modem statement*⟩ for the ⟨*line definition*⟩. A syntax error is generated, however, if **DIRECT** is named and a ⟨*line modem statement*⟩ is present.

Pragmatics

LINE ADAPTERS AND ADAPTER CLASSES

The line adapters are divided into eight "Adapter Classes." Refer to Table 5-3 for the Adapter Classes. The Touch-Tone®, Audio-Response, and Auto Dial Out line adapters comprise Adapter Classes 6, 7, and 8, respectively. The remaining line adapters comprise Adapter Classes 1 through 5. Adapter Classes 1 through 5 differ primarily in the maximum transmission speed at which the line adapters may be operated.

®Registered service mark of AT&T.

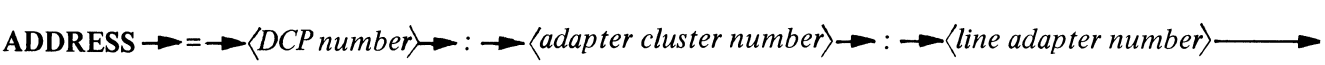
Table 5-3. Types of Line Adapters

Line Adapter	CLASS
ACII Adapter-Direct Connect Maximum transmission speeds:	
600 BPS	1
1800 BPS	2
2400 BPS	3
4800 BPS	4
9600 BPS	5
ACII Adapter-Data Set Control (Modem) Maximum transmission speeds:	
600 BPS	1
1800 BPS	2
2400 BPS	3
4800 BPS	4
9600 BPS	5
ACII Adapter-BDI (Burroughs Direct Interface) Connect Maximum transmission speeds:	
600 BPS	1
1800 BPS	2
2400 BPS	3
4800 BPS	4
9600 BPS	5
For Touch-Tone® telephone input	6
For Audio-Response line	7
ACII Adapter - Auto Dial Out	8

®Registered service mark of AT&T.

LINE ADDRESS STATEMENT

Syntax



Example

```
ADDRESS = 2:0:15.
```

The above example would appear in the ⟨*line definition*⟩ of the line at the 15th line adapter position in adapter cluster number 0 of DCP number 2.

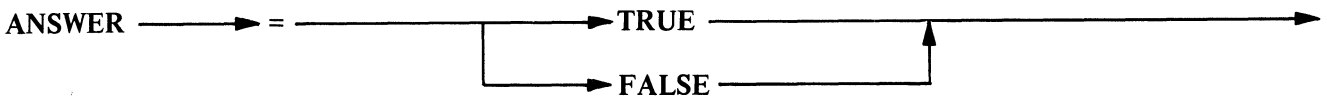
Semantics

The ⟨*line address statement*⟩ identifies the DCP number, the adapter cluster number, and the line adapter number of the defined logical line. If two DCPs share hardware-exchanged adapter clusters (as defined by the ⟨*DCP exchange statement*⟩ in a ⟨*DCP definition*⟩), then the ⟨*DCP number*⟩ defined in this statement is the DCP initially expected to service the adapter cluster of which the line is a part. This statement, which is required, must be defined explicitly in each ⟨*line definition*⟩.

Line Answer Statement

LINE ANSWER STATEMENT

Syntax



Semantics

The ⟨*line answer statement*⟩ defines whether or not (**TRUE** or **FALSE**, respectively) the DCP is to automatically answer an incoming call. This statement is required if the ⟨*line type statement*⟩ in the ⟨*line definition*⟩ defines the line configuration as **DIALIN** only, or **DIALIN** and **DIALOUT**.

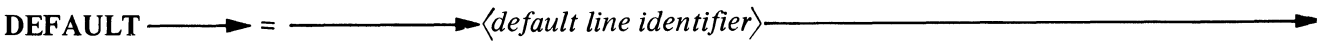
If **ANSWER = FALSE**, an incoming call causes the following actions to be taken by the DCP. A SWITCHED STATUS RESULT (CLASS = 7) message is sent to the MCS of the station that is the first entry in the Line Table for that line. (Unless an MCS has reconfigured the line so that it changes the first entry, the first entry in the Line Table will be the entry for the first station listed in the ⟨*line station statement*⟩ of the ⟨*line definition*⟩.) The message has a bit set in it that indicates the line is in a "ringing" status. Presumably, upon notification of a line in a ringing status, the MCS programmer instructs the DCP to answer the phone, or it takes appropriate action to clear the line.

If **ANSWER = TRUE**, an incoming call causes the DCP to take the following actions. A SWITCHED STATUS RESULT (CLASS = 7) message is sent to the controlling MCS of the station that is the first entry in the Line Table for that line. In this case the message has a bit set indicating that there has been an incoming call, and that the DCP is in the process of answering the call.

An MCS may change the value of **ANSWER** after DCP initialization, by means of a SET/RESET AUTO-ANSWER (TYPE = 102) DCWRITE.

LINE DEFAULT STATEMENT

Syntax



Example

```
DEFAULT = DFLTLIST1.
```

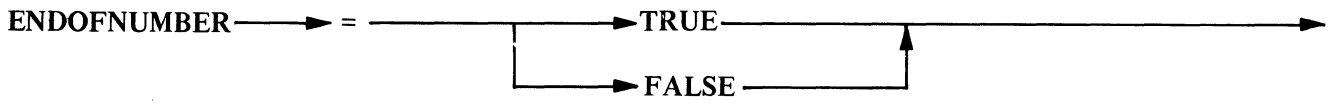
Semantics

The ⟨*line default statement*⟩ allows the programmer to specify the ⟨*default line identifier*⟩ of a set of default line attributes to be used for a ⟨*line definition*⟩ whose description is incomplete. It is advantageous to group attributes that several lines have in common under a Default ⟨*line definition*⟩ and list the remaining attributes under each individual ⟨*line definition*⟩ . The compiler will then refer to the Default ⟨*line definition*⟩ to complete the ⟨*line definition*⟩. The ⟨*line default statement*⟩ is not required to appear in a ⟨*line definition*⟩ ; however, a ⟨*line definition*⟩ must define all required attributes if a ⟨*line default statement*⟩ does not appear.

The ⟨*line default statement*⟩ can appear in a ⟨*line definition*⟩ or a Default ⟨*line definition*⟩. Thus, ⟨*line default statement*⟩*s* can be "nested" to combine the attributes of one or more Default ⟨*line definitions*⟩*s*.

LINE ENDOFNUMBER STATEMENT

Syntax



Semantics

The ⟨*line endofnumber statement*⟩ applies only to ⟨*line definition*⟩*s* that specify the Automatic Calling Unit (ACU) Adapter Class in its ⟨*line adapter class statement*⟩ (e.g., **ADAPTER = 8**). This statement is required for those ⟨*line definition*⟩*s*, and specifies whether or not (**TRUE** or **FALSE**, respectively) the ACU has an "end of number" option.

LINE MAXSTATIONS STATEMENT

Syntax

Example

```
MAXSTATIONS = 25.
```

Semantics

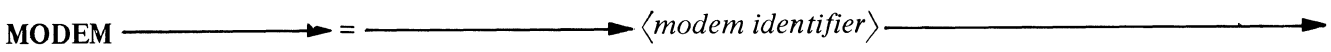
The ⟨*line maxstations statement*⟩ specifies the number of stations that may be assigned to the defined line. If this statement does not appear for a line having assigned stations (the ⟨*line station statement*⟩ lists all stations initially assigned to a line), it is assumed that **MAXSTATIONS** is the number of stations explicitly specified as assigned to the line in the ⟨*line station statement*⟩. The ⟨*integer*⟩ specified must not equal 0, exceed 255, or be less than the number of stations listed in the ⟨*line station statement*⟩ (if the ⟨*line station statement*⟩ is defined).

Pragmatics

This statement informs the compiler of the maximum number of station descriptors required in the Line Table of the DCP's table structure. By defining **MAXSTATIONS** to be greater than the number of stations listed in the ⟨*line station statement*⟩, an MCS may reconfigure more stations onto the line at some point in time after DCP initialization. For information regarding reconfiguration, refer to the B 5000/B 6000/ B 7000 Series DCALGOL Reference Manual.

LINE MODEM STATEMENT

Syntax



Example

```
MODEM = BELL103A.
```

Semantics

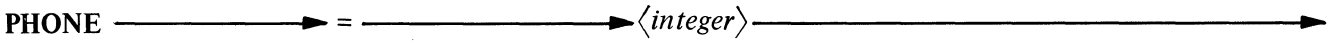
The ⟨*line modem statement*⟩ specifies the modem type that exists on the system end of the physical line. (The ⟨*station modem statement*⟩ in a ⟨*station definition*⟩ specifies the modem type connected to the line on the terminal end.)

Pragmatics

The compiler references other portions of the program with this statement, checking for consistency. If, for example, the ⟨*modem definition*⟩ of the ⟨*modem identifier*⟩ specified in this statement lists any ⟨*communication type number*⟩ *s* in its ⟨*modem adapter statement*⟩ that are not compatible with the Adapter Class specified in the ⟨*line adapter class statement*⟩ of the ⟨*line definition*⟩, then a syntax error is generated. Another situation that causes a syntax error to be generated is if the compiler discovers that the modem type specified in this statement is not compatible, in respect to the ⟨*communication type number*⟩, with the modem type specified in the ⟨*station definition*⟩ of a station assigned to the line.

LINE PHONE STATEMENT

Syntax



Example

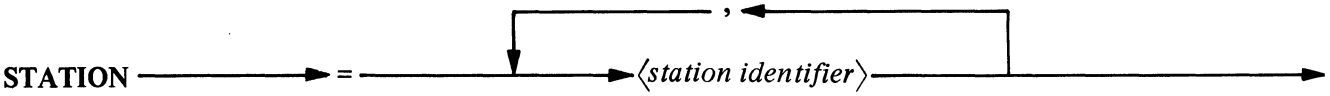
PHONE = 12136572385.

Semantics

The ⟨*line phone statement*⟩, implemented for documentation purposes only, documents the telephone number of a **DIALIN** type line. This statement is optional in a ⟨*line definition*⟩.

LINE STATION STATEMENT

Syntax



Examples

```
STATION = RJE1.
STATION = DAKOTA/KID, BIDS.
```

Semantics

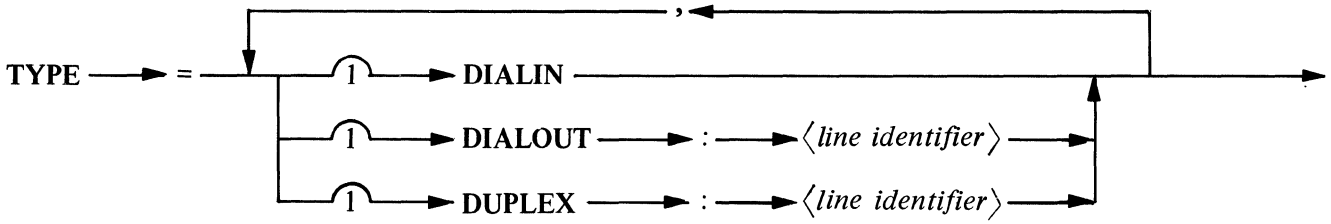
The ⟨*line station statement*⟩ is the means by which the NDL programmer associates one or more stations with a line. A station that is associated with a particular line is said to be "assigned" to that line.

This statement is required in those ⟨*line definition*⟩ *s* that specify **DUPLEX** in the ⟨*line type statement*⟩. In all other variations of ⟨*line type statement*⟩, this statement is optional.

If more than one station is named, each station must have the same ⟨*communication type number*⟩ defined in its respective ⟨*station adapter statement*⟩.

LINE TYPE STATEMENT

Syntax



Examples

```
TYPE = DIALIN.
TYPE = DIALOUT:ACULINE.
TYPE = DUPLEX:AUXLINE.
TYPE = DIALIN, DIALOUT:AUTOCALL, DUPLEX:SUPERVISORY.
```

Semantics

The ⟨*line type statement*⟩ provides the compiler with specific information concerning special logical line configurations. This statement is required for ⟨*line definition*⟩*s* whose line utilize either dial-in, dial-out, or full duplex hardware facilities.

DIALIN

This form identifies the line as a dial-in line. A line that may be dialed from a remote site is a dial-in line. The appropriate ⟨*line type statement*⟩ for this configuration would be:

```
TYPE=DIALIN.
```

A logical line defined in this manner must include the ⟨*line answer statement*⟩ and the ⟨*line modem statement*⟩. The ⟨*line definition*⟩ for such a line could appear as follows:

```
LINE DIALUPLINE:
    TYPE     = DIALIN.
    ADDRESS  = 0:0:0.
    MODEM    = TTY103A.
    STATION  = DIALUPSTATION.
    ANSWER   = TRUE.
    ADAPTER  = 1(MODEM).
```

DIALOUT

This form identifies the line as a dial-out line. A dial-out line is defined as a line that can become connected to a remote site as a result of a Message Control System issuing a DIALOUT (TYPE = 98) DCWRITE to the line (thereby causing an Automatic Calling Unit (ACU) to dial the phone number of the remote site). The **TYPE=DIALOUT:** ⟨*line identifier*⟩ syntax of the statement specifies such a configuration. The ⟨*line identifier*⟩ names the ⟨*line definition*⟩ that defines the associated ACU. The following example illustrates how the ⟨*line definition*⟩ *s* could appear for a dial-out configuration.

```
LINE DIALOUTLINE:
    TYPE        = DIALOUT:ACULINE.
    ADDRESS     = 0:0:1.
    MODEM       = TTY103A.
    ADAPTER     = 1(MODEM).
```

```
LINE ACULINE:
    ENDOFNUMBER  = FALSE.
    ADDRESS      = 0:0:5.
    ADAPTER      = 8.
```

The ⟨*line definition*⟩ for the dial-out line must include a ⟨*line modem statement*⟩ and cannot include a ⟨*line station statement*⟩. The ⟨*line definition*⟩ for the ACU must include a ⟨*line endofnumber statement*⟩, and it must define an address (in the ⟨*line address statement*⟩) that is on the same adapter cluster as the associated dial-out line.

DUPLEX

This form identifies the line as the primary of a line pair, for purposes of simultaneous transmission and receptions. The ⟨*line identifier*⟩ names the auxiliary line's ⟨*line definition*⟩. The line referenced as the auxiliary cannot contain a ⟨*line type statement*⟩ nor a ⟨*line station statement*⟩.

The following is an example of how full duplex primary and auxiliary lines could be defined.

```
LINE DUPLEXPRIMARY:
    TYPE     = DUPLEX:DUPLEXAUXILIARY.
    ADDRESS  = 0:0:5.
    MODEM    = SUPERMODEM.
    STATION  = MODEL37.
    ADAPTER  = 1.

LINE DUPLEXAUXILIARY:
    ADDRESS  = 0:0:6.
    ADAPTER  = 1.
```

Pragmatics

COMBINED CONFIGURATIONS

A dial-in/dial-out line is characterized by both the ability to be dialed from a remote site, and the ability to become connected to a remote site as a result of a Message Control System issuing a DIALOUT (TYPE = 98) DCWRITE. This type of configuration requires the **DIALIN** and **DIALOUT**: ⟨*line identifier*⟩ options to appear in the ⟨*line type statement*⟩. The following example illustrates how a dial-in and dial-out ⟨*line definition*⟩ could appear:

```
LINE IOLINE:
    TYPE         = DIALIN,DIALOUT:AUTOCALLUNIT.
    ADDRESS      = 0:1:0.
    MODEM        = TTY103A.
    STATION      = REMOTETTY.
    ANSWER       = TRUE.
    ADAPTER      = 1(MODEM).

LINE AUTOCALLUNIT:
    ENDOFNUMBER  = FALSE.
    ADDRESS      = 0:1:1.
    ADAPTER      = 8.
```

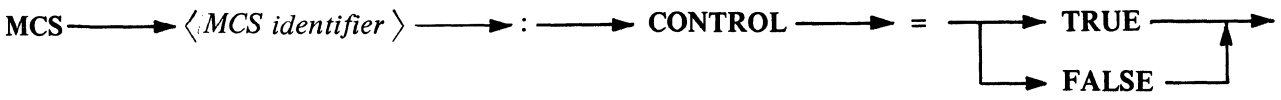
Line Type Statement – Continued

The full duplex syntax could be combined with the dial-in and dial-out syntax as follows:

```
LINE IODUPLEX:
    TYPE          = DIALIN,DIALOUT:AUTOCALLUNIT,DUPLEX:AUXLINE.
    ADDRESS       = 0:2:0.
    MODEM         = SUPERMODEM.
    STATION       = REMOTEDUPLEXDEVICE.
    ANSWER        = TRUE.
    ADAPTER       = 1(MODEM).

LINE AUXLINE:
    ADDRESS       = 0:2:1.
    ADAPTER       = 1(MODEM).

LINE AUTOCALLUNIT:
    ENDOFNUMBER   = FALSE.
    ADDRESS       = 0:2:2.
    ADAPTER       = 8.
```

MCS DEFINITION

Syntax



Examples

```
MCS SYSTEM/CANDE:CONTROL = FALSE.
MCS SYSTEM/DIAGNOSTICMCS. ON SIXPACK:CONTROL = TRUE.
```

Semantics

The purpose of the ⟨*MCS definition*⟩ is twofold: First, the ⟨*MCS definition*⟩ adds the ⟨*MCS identifier*⟩ to the list (contained in the Network Information File) of valid Message Control System (MCS) programs; and second, the ⟨*MCS definition*⟩ specifies whether or not (**CONTROL = TRUE**, or **CONTROL = FALSE**, respectively) the named MCS is allowed to execute a limited set of DCWRITE functions that perform DCP diagnostic functions in addition to the standard DCWRITEs. ⟨*MCS identifier*⟩ has the syntactic form of a ⟨*system identifier*⟩.

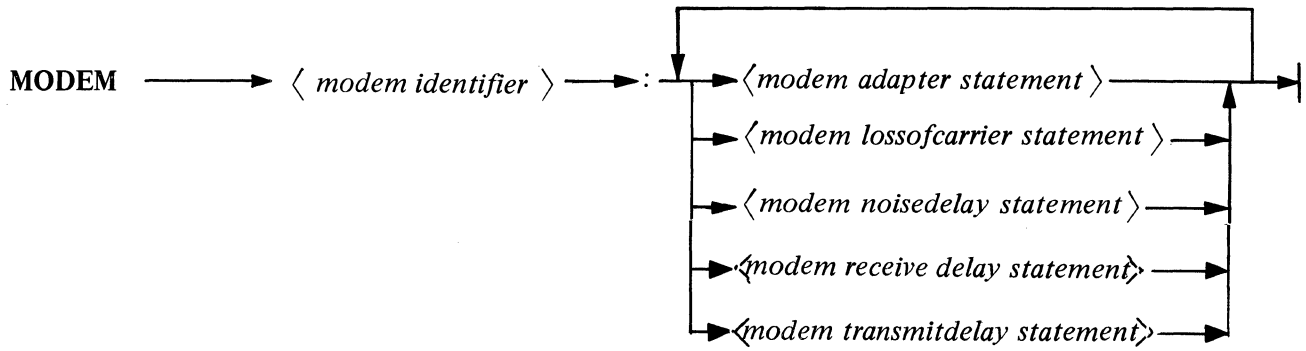
Pragmatics

A list of valid MCSs is maintained in the Network Information File in order to restrict unauthorized DCALGOL programs from becoming an MCS. (A DCALGOL program becomes an MCS when it successfully executes an INITIALIZE PRIMARY QUEUE (TYPE = 0) DCWRITE.) The MCS declaration is one means of adding a name of an MCS to that list. (One other means is the ⟨*station MCS statement*⟩ in a ⟨*station definition*⟩.)

The diagnostic DCWRITE functions allow an MCS to perform on-line tests of components in the Data Communications System. Those DCWRITEs that may be utilized in an MCS when **CONTROL = TRUE** have DCWRITE TYPE numbers greater than 159.

MODEM DEFINITION

Syntax



Example

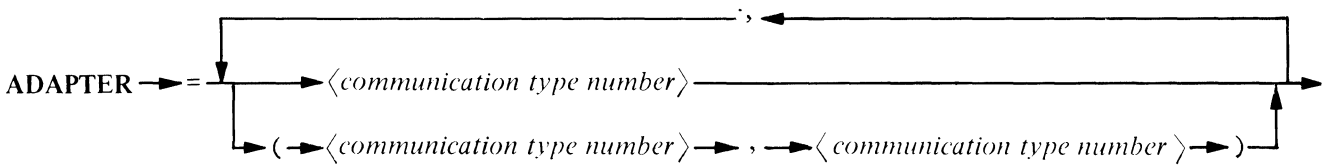
```
MODEM MABELL103A:
    ADAPTER = 4.
    LOSSOFCARRIER = DISCONNECT.
    NOISEDELAY = 0.
    TRANSMITDELAY = 0.
```

Semantics

The ⟨*modem definition*⟩ defines the attributes of a modem type in the data communications system. The ⟨*modem identifier*⟩ names the ⟨*modem definition*⟩, and has the syntactic form of ⟨*identifier*⟩. The ⟨*modem statement*⟩*s* are described subsequently.

MODEM ADAPTER STATEMENT

Syntax



Examples

```
ADAPTER = 1,2,3,4.
ADAPTER = 10.
ADAPTER = (2,3), (4,5), 6,7.
```

Semantics

The ⟨*modem adapter statement*⟩ defines one or more combinations of character format, synchronous/asynchronous communication, and line speed (in the case of asynchronous communications) with which the modem is compatible. This is done by supplying one or more ⟨*communication type number*⟩*s* (or number pairs).

Table 5-4 lists the allowed ⟨*communication type number*⟩*s* and the characteristics associated with each. For example, the statement

```
ADAPTER = 4.
```

defines an 11-bit character format, asynchronous communication, at a line speed of 110 bits per second.

If the modem is to be used in a full duplex mode, and the primary and auxiliary lines have different characteristics, then one or more ⟨*communication type number*⟩ pairs must be supplied. For example, the statement

```
ADAPTER = (11,6).
```

defines for the primary line a 10-bit character format, synchronous communication, at a speed of 1800 bits per second. The characteristics associated with the auxiliary line are the same as for the primary line, except that the auxiliary line runs at a line speed of 150 bits per second.

Pragmatics

COMMUNICATION TYPE NUMBERS

A ⟨*communication type number*⟩ is an integer that has associated with it a set of attributes that define three line characteristics. Those characteristics are the format of the characters transmitted (start information, data information, parity information, and stop information), whether the line is to be driven synchronously or asynchronously, and the speed of the transmissions (in the case of asynchronous communications). Table 5-4 lists the allowed ⟨*communication type number*⟩*s* and the line characteristics associated with each.

Most of the electronics that directly control a line are located in the adapter cluster that contains the line adapter for that line (rather than being located in the line adapter itself). The adapter cluster is somewhat general purpose in its design in that it can run at various line speeds and handle various character formats. The DCP can cause the adapter cluster to function in a suitably special-purpose way (with respect to a single line) by supplying it a number derived from the ⟨*communication type number*⟩.

There are three areas in an NDL program that require the programmer to supply one or more ⟨*communication type number*⟩*s*:

a. In the ⟨*modem adapter statement*⟩ of each ⟨*modem definition*⟩,

b. In the ⟨*terminal adapter statement*⟩ for each ⟨*terminal definition*⟩, and

c. In the ⟨*station adapter statement*⟩ for each ⟨*station definition*⟩.

As it encounters each area, the NDL compiler cross-checks to determine if the areas are compatible in their description. If inconsistencies in component compatibility arise, syntax errors are generated. Restrictions are described in the ⟨*terminal adapter statement*⟩ and ⟨*station adapter statement*⟩ semantics.

EXPLANATION OF TABLE 5-4

Table 5-4 lists the allowed ⟨*communication type number*⟩*s* in the column labeled "COMM. TYPE NUM." To the right of each ⟨*communication type number*⟩ are the three line characteristics associated with it, under the columns labeled "SPEED (BPS)," "CHARACTER FORMAT," and "SYNCHRONOUS OR ASYNCHRONOUS." The rightmost column, labeled "COMPATIBLE ADAPTER CLASSES," is referenced and described in the ⟨*line definition*⟩ section of this chapter.

Table 5-4. Table of ⟨*communication type number*⟩*s*

COMM. TYPE NUM.	SPEED (BPS)	CHAR. SIZE (BITS)	CHARACTER FORMAT				SYNCHRONOUS OR ASYNCHRONOUS	COMPATIBLE ADAPTER CLASSES
			START INFO.	DATA INFO.	PARITY INFO.	STOP INFO.		1 2 3 4 5 6 7 8
1	45.5	7.5	1	5.0	—	1.5	ASYNC.	X X X X X
2	56.9	7.5	1	5.0	—	1.5	ASYNC.	X X X X X
3	75.0	7.5	1	5.0	—	1.5	ASYNC.	X X X X X
4	110.0	11.0	1	7.0	1	2	ASYNC.	X X X X X
5	134.5	9.0	1	6.0	1	1	ASYNC.	X X X X X
6	150.0	10.0	1	7.0	1	1	ASYNC.	X X X X X
7	300.0	10.0	1	7.0	1	1	ASYNC.	X X X X X
8	600.0	10.0	1	7.0	1	1	ASYNC.	X X X X X
9	1200.0	10.0	1	7.0	1	1	ASYNC.	X X X X
10	1200.0	6.0	1	4.0	—	1	ASYNC.	X X X X
11	1800.0	10.0	1	7.0	1	1	ASYNC.	X X X X
12	2400.0	10.0	1	7.0	1	1	ASYNC.	X X X
13	3600.0	10.0	1	7.0	1	1	ASYNC.	X X
14	4800.0	10.0	1	7.0	1	1	ASYNC.	X X
15	9600.0	10.0	1	7.0	1	1	ASYNC.	X
16	2000.0	7.0	—	6.0	1	—	SYNC.	X X X
17	2000.0	8.0	—	7.0	1	—	SYNC.	X X X
18	2000.0	9.0	—	8.0	1	—	SYNC.	X X X
19	2400.0	7.0	—	6.0	1	—	SYNC.	X X X
20	2400.0	8.0	—	7.0	1	—	SYNC.	X X X
21	2400.0	9.0	—	8.0	1	—	SYNC.	X X X
22	4800.0	7.0	—	6.0	1	—	SYNC.	X X
23	4800.0	8.0	—	7.0	1	—	SYNC.	X X
24	4800.0	9.0	—	8.0	1	—	SYNC.	X X
25	9600.0	7.0	—	6.0	1	—	SYNC.	X
26	9600.0	8.0	—	7.0	1	—	SYNC.	X
27	9600.0	9.0	—	8.0	1	—	SYNC.	X
28	40.0	4.0	—	4.0	—	—	SYNC.	X
29	16.0	8.0	—	8.0	—	—	SYNC.	X
30	40.0	4.0	—	4.0	—	—	SYNC.	X

MODEM LOSSOFCARRIER STATEMENT

Syntax

Pragmatics

Certain modems (Western Electric (Bell System) 103 series modems, and possibly others) maintain continuous carrier in both directions while the line is properly connected. As such, CF (Carrier Detected) and CB (Clear to Send) are maintained TRUE while connected. Additionally, if each modem is equipped with both the Initiate Disconnect and the Respond to Disconnect options, each modem employs the "long space disconnect" convention. This convention allows one modem to determine if the other is disconnecting, and itself go "on-hook" and drop CC (Data Set Ready).

Two problems arise, however, when only one such modem is configured at the system end, and the terminal is interfaced with an acoustic coupler at the terminal end. At the time of making a connection, establishment of carrier is difficult. In fact, the system modem may detect carrier from the coupler while the telephone receiver is near the coupler and before the receiver is properly seated. In this case, CF and CB are raised prematurely, and if the system takes this as a cue to begin transmission of a greeting, the two signals (the data transmitted from the system, and carrier from the acoustic coupler) interact with each other, and the system modem detects loss of carrier. At the time of terminating a call, if the terminal initiates the disconnect and has no "long space disconnect" facility, or if the terminal operator does not use it, the system modem detects only loss of carrier. In this case, the system modem drops CF and CB, the modem remains "off-hook" and maintains CC (Data Set Ready). Thereafter, any incoming calls would receive a "busy" signal.

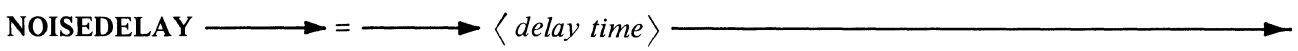
The ⟨*modem lossofcarrier statement*⟩ is implemented for such a configuration. If this statement is included in the definition of a modem, special logic is invoked, in addition to the normal logic, when dealing with that modem type.

In the case of the system calling out, normal logic waits for CC to be raised by the modem. If CC is raised within 25 seconds, the line is immediately released as connected. A timeout of 25 seconds causes CD (Data Terminal Ready) to be dropped, the modem goes "on-hook," and the line reverts to a disconnected state. The special logic is invoked after CC is found TRUE. With the 25-second timeout in effect, the special logic then waits until CF and CB are both raised by the modem. After CF and CB are detected, the logic then delays approximately 5 seconds before notifying the system that the line is connected. This gives the terminal operator sufficient time to place the receiver in the acoustic coupler.

In the case of a terminal-initiated disconnect, that condition is detected in the normal logic by either the "long space disconnect" adapter cluster interrupt or by a CC (Data Set Ready) FALSE condition. In addition to the normal logic, the special logic also interprets CF FALSE or CB FALSE as a terminal-initiated disconnect.

MODEM NOISEDELAY STATEMENT

Syntax



Examples

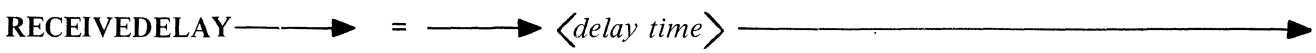
```
NOISEDELAY = 0.
NOISEDELAY = 200 MILLI.
```

Semantics

The ⟨*modem noisedelay statement*⟩ defines the amount of time that should be delayed when the modem enters a Clear to Send (CB) status to avoid receiving "noise" on the line. ⟨*delay time*⟩ must be expressed as ⟨*time*⟩, and affects the amount of time delayed after an **INITIATE RECEIVE** or **INITIATE TRANSMIT** construct is executed and before the next statement is executed in a ⟨*control definition*⟩ or ⟨*request definition*⟩. The ⟨*delay time*⟩ defined in this statement is used in a compiler algorithm that calculates the delay. The compiler algorithm is discussed in the semantics of the **INITIATE RECEIVE** and **INITIATE TRANSMIT** constructs under the ⟨*initiate statement*⟩. This statement must appear in each ⟨*modem definition*⟩.

MODEM RECEIVEDELAY STATEMENT

Syntax



The ⟨*modem receivedelay statement*⟩ defines the amount of time a data carrier signal (carrier) must remain present at the receiving modes above receiver threshold signal strength before carrier (CF) is enabled and receive data (BB) is unclamped; i.e. the amount of time before the modem is considered to be in a valid receive state.

If this statement is referenced in a given modem definition, the ⟨*modem noisedelay statement*⟩ for that modem then defines the amount of additional time required for the modem to suppress echoes and filter spurious signals before the presence of valid data can be assumed on the receive data line (BB).

The ⟨*modem transmitdelay statement*⟩ defines the amount of time the modem requires to switch to a Clear-to-Send (CB) state after receiving a Request-to-Send (CA). This delay is called the "Request-to-Send--Clear-to-Send Delay."

STATION delays for a station that references a modem with a specified receive delay are computed by the following algorithms:

Receive delay : : = modem receive delay + modem noise delay

Transmit delay : : = **MAX**(modem transmit delay, terminal turnaround)

Examples

```
MODEM    TA713:
         TRANSMITDELAY  = 20 MILLI.
         NOISEDELAY     = 0.
         RECEIVEDELAY   = 0.
         ADAPTER        = 4,5,6,7,8,9,10.

MODEM    M208S:
         TRANSMITDELAY  = 170 MILLI.
         RECEIVEDELAY   =  51 MILLI.
         NOISEDELAY     =   0.
         ADAPTER        = 23.

MODEM    M202C:
         TRANSMITDELAY  = 225 MILLI.
         RECEIVEDELAY   =  50 MILLI.
         NOISEDELAY     = 170 MILLI.
         ADAPTER        = 4,5,6,7,8,9,10.
```

For the modems in the examples above, **STATION** delays computed for stations referencing the respective modems with a terminal turnaround time less than the modem transmit delay would be:

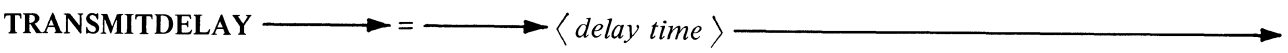
MODEM TA713
Transmit delay : : = 20 milliseconds
Receive delay : : = 0 microseconds

MODEM M208S
Transmit delay : : = 170 milliseconds
Receive delay : : = 51 milliseconds

MODEM M202C
Transmit delay : : = 225 milliseconds
Receive delay : : = 220 milliseconds

MODEM TRANSMITDELAY STATEMENT

Syntax



Examples

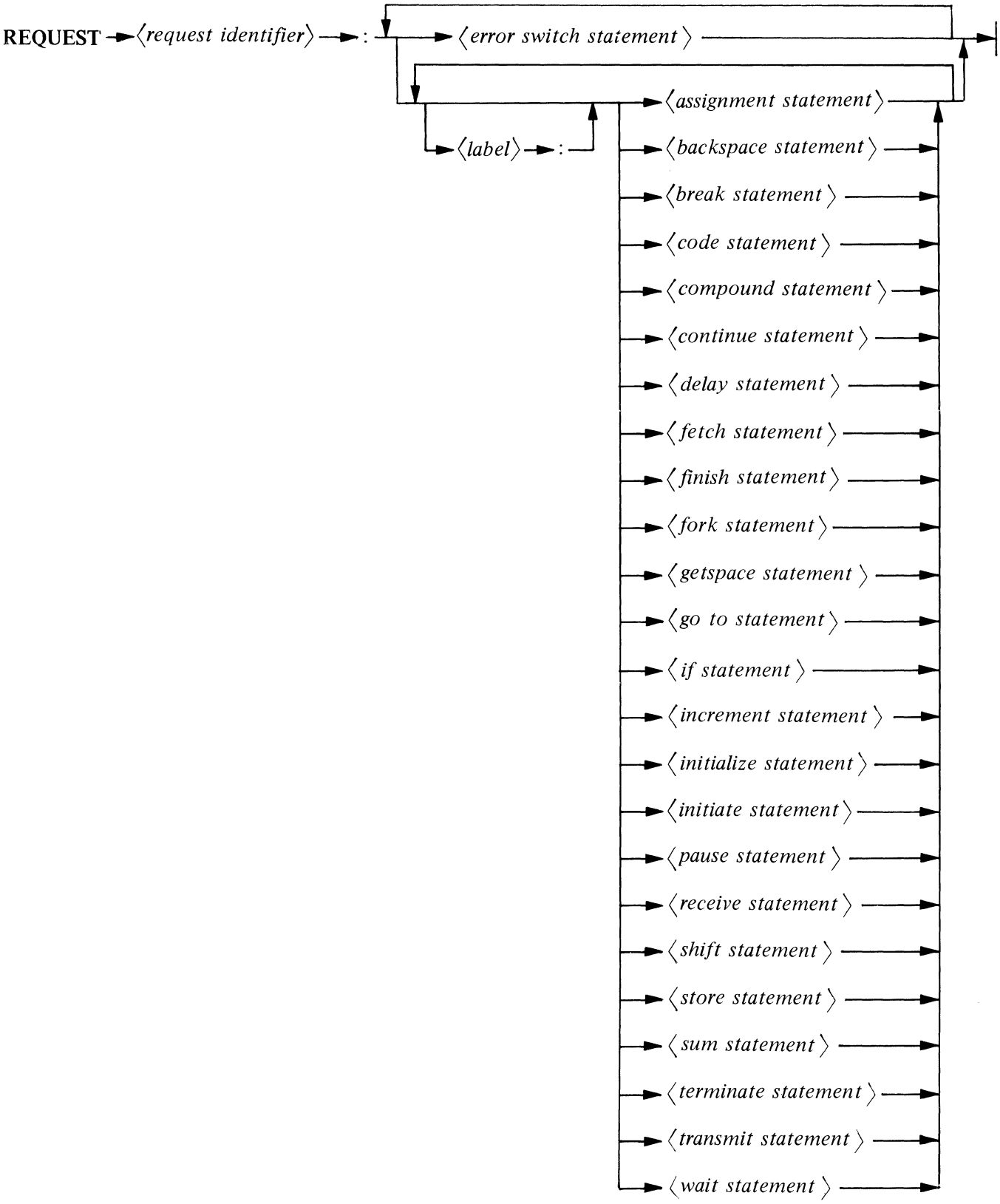
```
TRANSMITDELAY = 0.
TRANSMITDELAY = 150 MICRO.
```

Semantics

The ⟨*modem transmitdelay statement*⟩ defines the amount of time required for the modem to switch to a Clear to Send (CB) state after receiving a Request to Send (CA). ⟨*delay time*⟩ must be expressed as ⟨*time*⟩ and affects the amount of time delayed after an **INITIATE RECEIVE** or **INITIATE TRANSMIT** construct is executed and before the next statement is executed in a ⟨*control definition*⟩ or ⟨*request definition*⟩. The ⟨*delay time*⟩ defined in this statement is used in a compiler algorithm that calculates the delay. The compiler algorithm is discussed in the semantics of the **INITIATE RECEIVE** and **INITIATE TRANSMIT** constructs of the ⟨*initiate statement*⟩. This statement must appear in each ⟨*modem definition*⟩.

REQUEST DEFINITION

Syntax



Example

```
REQUEST READTTY:

    INITIATE RECEIVE.
    RECEIVE TEXT [END].
    TERMINATE NORMAL.
```

Semantics

⟨*request definition*⟩*s*, sometimes referred to as Requests, are coded line disciplines (protocols) that are used in communicating with the various terminal types in the data communications network. A ⟨*request definition*⟩ must be coded for each capability of a terminal type; if it is possible for a terminal type to send input to the system and receive output from the system, then two ⟨*request definition*⟩*s* must be specified for that terminal type in its ⟨*terminal definition*⟩. The input ⟨*request definition*⟩ is generally referred to as the "Receive Request," and the output ⟨*request definition*⟩ the "Transmit Request." (The specific ⟨*request definition*⟩ to be used for each of these capabilities is specified by the ⟨*terminal request statement*⟩.

When there is a message to be sent to a particular station on a line, the ⟨*control definition*⟩ initiates the Transmit Request specified for the ⟨*terminal definition*⟩ associated with the station. The Transmit Request procedure handles the transmission of the message. If the transmission of the message is successful, the Transmit Request is terminated, and a branch of control is made back to the ⟨*control definition*⟩ for the initiation of the next Request.

If the terminal associated with a station is allowed to input data, the ⟨*control definition*⟩ designated for that line normally initiates the Receive Request specified for the terminal type. If the terminal has information to transmit, the Receive Request procedure obtains a message space in which to store the received text, receives and stores the text, and then terminates in a manner that forwards the message to the MCS. If the terminal has nothing to transmit, the Receive Request procedure usually notes that there was no input, and terminates. In either case, upon termination, control returns to the ⟨*control definition*⟩ for the initiation of the next Request.

⟨*request identifier*⟩ has the syntactic form of ⟨*identifier*⟩.

Statements in ⟨*request definition*⟩*s* are executed sequentially. In some cases, however, it is desirable to alter the order of execution of statements within the procedure. A ⟨*request statement*⟩ preceded by a ⟨*label*⟩ is one means of accomplishing this. The ⟨*go to statement*⟩ is used to transfer control to a ⟨*label*⟩*ed* ⟨*request statement*⟩

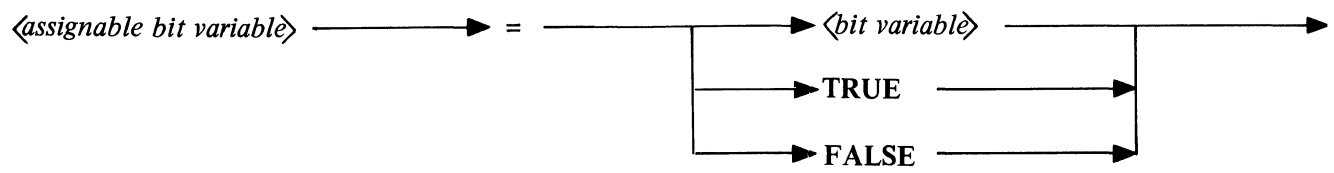
A ⟨*request statement*⟩ must be appropriate for the type of Request in which it appears. That is, some ⟨*request statement*⟩*s* are allowed only in Receive Requests, some are allowed only in Transmit Requests, and some are allowed in either type. Subsequently, the semantics portion of each statement defines, among other things, in which type of ⟨*request definition*⟩ the statement can appear.

Due to the table driven implementation in the **DCP**, system data and code are separated for a given request-set, thus allowing only one representation of the request-set for a variety of stations. However, if stations differ in certain ways (such as **TERMINAL TURNAROUND, PARITY, CODE**) that are not part of the table description, then such data will be incorporated in the code. Careful consideration must be applied to such attributes as they will cause multiple-representations of the code.

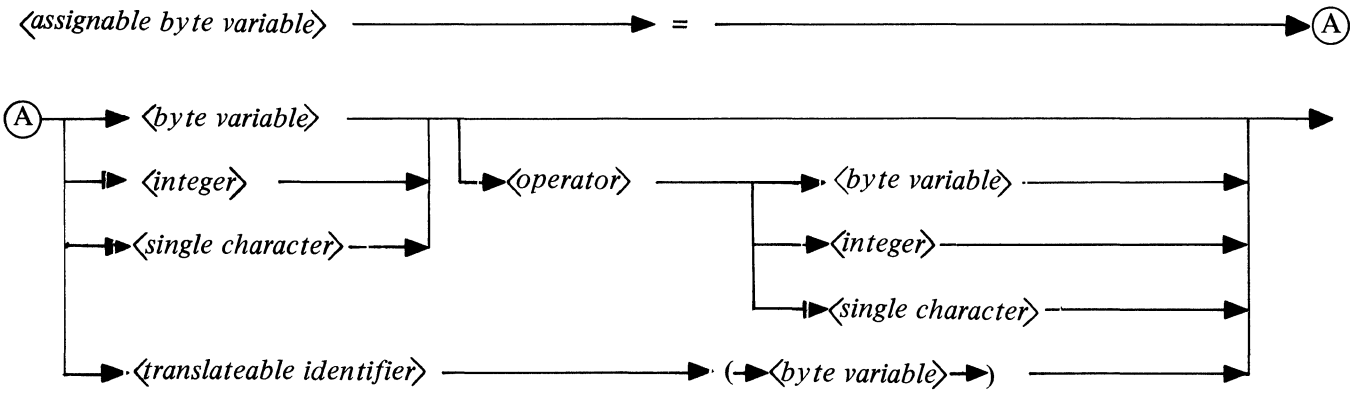
ASSIGNMENT STATEMENT

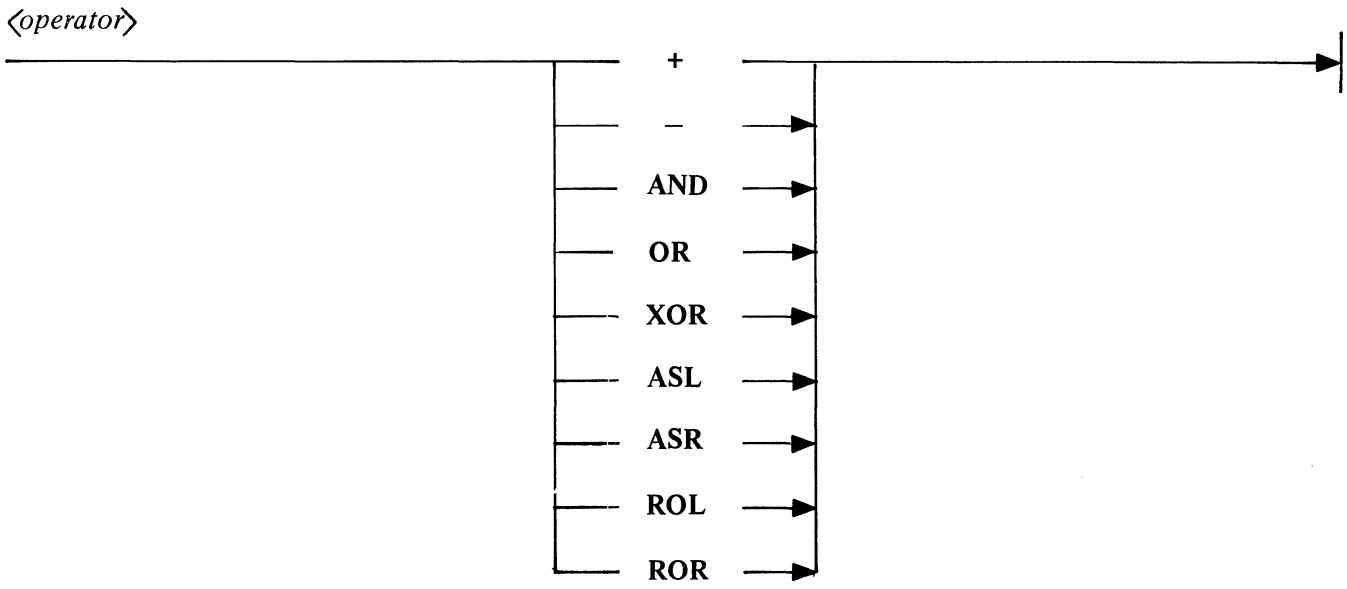
Syntax

FORM 1 – LOGICAL ASSIGNMENT



FORM 2 – VALUE ASSIGNMENT





Examples

```
TOG[0] = TRUE.
TOG[1] = TOG[0].
LINE(BUSY) = FALSE.
TALLY[2] [5] = TRUE.
TOG[3] = TALLY[0] [7].
TOG[6] = CHAR[4].
STATION = MAXSTATIONS.

TALLY[0] = STATION(FREQUENCY) – TALLY[1].
CHARACTER = TRANSTABLEID(CHARACTER).
STATION = RECEIVE ADDRESS(TRANSMIT)[ADDERR:999].

AI = CHARACTER ASR 3.
TALLY[2] = TALLY[0] ASL 1.
LINE(TALLY[1]) = STATION(FREQUENCY) ROL RETRY.

CHARACTER = CHARACTER ROR 6
TALLY [0] = CHARACTER AND 4 "03".
```

Semantics

FORM 1

This form causes the value on the right side of the equal sign to replace the current value of ⟨*assignable bit variable*⟩.

FORM 2

Value assignment causes a calculated value on the right of the equal sign to be stored in the ⟨*assignable byte variable*⟩.

Arithmetic calculations (+ and -) are done in modulo 256 arithmetic.

The logical operators **AND**, **OR**, and **XOR** are defined by Table 5-0.

Table 5-5. Truth Table

BIT A	BIT B	A AND B	A OR B	A XOR B
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	FALSE

The logical operations defined above are performed on all 8 bits of the operands (⟨*byte variables*⟩s, ⟨*integers*⟩s, and/or ⟨*single characters*⟩s).

The shift operators (**ASL**, **ASR**, **ROL**, **ROR**) allow for "shifting" the contents of byte variables or constants.

The **ASL** and **ASR** operators are arithmetic shifts, to the left and right, respectively. The first operand is shifted by a number of bits specified by the second operand, with zeroes being shifted into the operand and bits being lost from the opposite end.

For example, if

TALLY[0] = 4.

then

TALLY[2] = TALLY[0] ASL 1.

would double the value of **TALLY[0]** by shifting it left one, thus storing the value, 8 in **TALLY[2]**

Similarly, if

CHARACTER = 4 "3C".

then

AI = CHARACTER ASR 3.

would shift the value 4 "3C" right by 3 bits, resulting in the value, 4 "07" being placed in the **AI** register.

The **ROL** and **ROR** operators are rotational shifts, with bits being shifted back into the operand from which they are lost.

Thus if

STATION(FREQUENCY) = 9.
RETRY = 6.

then

LINE(TALLY[1]) = STATION(FREQUENCY) ROL

would cause the contents of **STATION(FREQUENCY)**, 9, to be rotated left 6 bits, the value of **RETRY**. Thus the value 4 "42" would be stored in **LINE(TALLY[1])**.

Likewise, if

CHARACTER = 4 "3C".

then

CHARACTER = CHARACTER ROR 6

would rotate the contents of **CHARACTER** right six bits. The resulting value, 4 "F0", is then being stored back in **CHARACTER**.

⟨*assignable byte variable*⟩ = ⟨*translatetable identifier*⟩ (⟨*byte variable*⟩).

This construct is the means to invoke user-defined character translation. User-defined translation is effected by three areas of the NDL source program.

a. In a ⟨*translatetable definition*⟩, the programmer must define the contents of a translation table and associate a ⟨*translatetable identifier*⟩ with it.

b. In the ⟨*terminal definition*⟩ of a terminal type that requires special character translation, the programmer should suppress automatic character translation by using either of the following forms of the ⟨*terminal code statement*⟩:

CODE = BINARY.

or

CODE = EBCDIC.

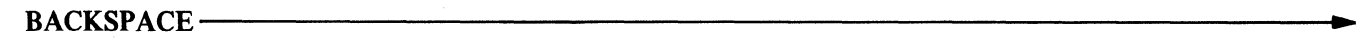
c. In a ⟨*control definition*⟩ or ⟨*request definition*⟩, the programmer invokes the translation by using this option of the value assignment. Any ⟨*byte variable*⟩ can be designated as containing the character to be translated.

The ⟨*translatetable identifier*⟩ identifies the translation table to be used. An ⟨*assignable byte variable*⟩ is designated to the left of the equal sign, identifying where the resulting translated character is to be stored.

If N is the ⟨*source size*⟩ (defined in the ⟨*translatetable definition*⟩), then the N low-order bits of the ⟨*byte variable*⟩ are used as an index into the translation table. The eight-bit character thus indexed is stored in the ⟨*assignable byte variable*⟩.

BACKSPACE STATEMENT

Syntax

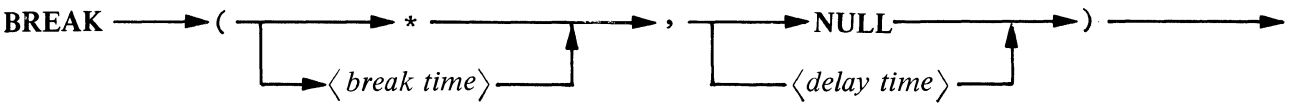


Semantics

The ⟨*backspace statement*⟩ causes the message text pointer to be moved backwards one character. This statement can only appear in a Receive Request. The ⟨*backspace statement*⟩ may be executed repeatedly; however, the message text pointer will never be stepped back so far that it points into the message header.

BREAK STATEMENT

Syntax



Examples

```
BREAK (*, NULL).
BREAK (200 MILLI, 3 SEC).
BREAK (*, 3 SEC).
BREAK (100 MILLI, NULL).
```

Semantics

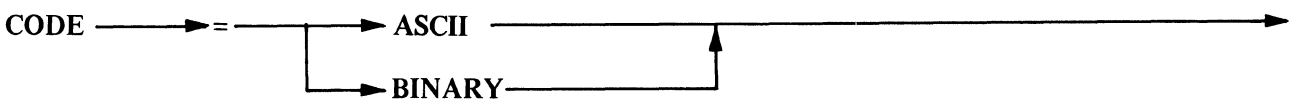
The ⟨*break statement*⟩ causes binary zeroes to be transmitted on the line, thus changing the state of the line to a "spacing" condition for a specified time.

The ⟨*break time*⟩ specifies the ⟨*time*⟩ to break. An asterisk indicates that a standard break of 2 character times should be used.

The ⟨*delay time*⟩ specifies the ⟨*time*⟩ to delay subsequent to the break and prior to when control continues.

CODE STATEMENT

Syntax



Semantics

CODE=ASCII invokes the ASCII-to-EBCDIC translation for received data and the EBCDIC-to-ASCII translation for transmitted data.

CODE=BINARY inhibits any character translation on data transmitted or received.

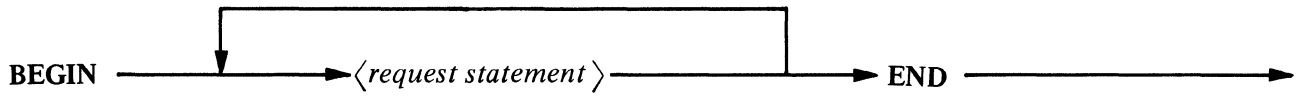
Pragmatics

The ⟨*code statement*⟩ allows a programmer to either invoke or inhibit on a logical line the DCP ASCII-to-EBCDIC character code translation for input, and the EBCDIC-to-ASCII character code translation for output. Any ⟨*terminal definition*⟩ that names, in its ⟨*terminal control statement*⟩, a ⟨*control definition*⟩ that utilizes the ⟨*code statement*⟩, must define **ASCII (BINARY)** as its character code in the ⟨*terminal code statement*⟩. (Refer to the ⟨*terminal code statement*⟩ in this chapter.)

Once that translation has been invoked on a line, the translation continues until such time that it is inhibited. If translation is inhibited, translation will be inhibited on that line until invoked again by the statement: **CODE ASCII**.

COMPOUND STATEMENT

Syntax



Example

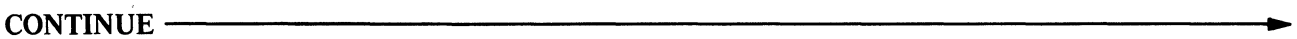
```
BEGIN
INITIATE TRANSMIT.
TRANSMIT TEXT.
FINISH TRANSMIT.
END.
```

Semantics

The ⟨*compound statement*⟩ groups several statements together to form a logical sequence. To execute more than one statement when the condition of an ⟨*if statement*⟩ is satisfied, a ⟨*compound statement*⟩ must be used.

CONTINUE STATEMENT

Syntax



Semantics

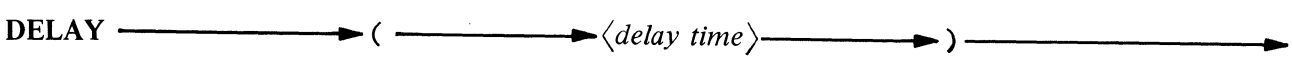
The ⟨*continue statement*⟩ can appear in only those ⟨*request definition*⟩ *s* and ⟨*control definition*⟩ *s* written to communicate with full duplex terminal types. This statement causes the co-line to resume processing, if, and only if, it had been suspended by a ⟨*wait statement*⟩, or a ⟨*receive statement*⟩ with a CONTINUE option specified. If the co-line had not been suspended, this statement acts as a no-op. The ⟨*continue statement*⟩ has no effect upon the line on which it was executed.

Pragmatics

Refer to the ⟨*fork statement*⟩ pragmatics.

DELAY STATEMENT

Syntax



Examples

```
DELAY (3 SEC).
DELAY (0).
```

Semantics

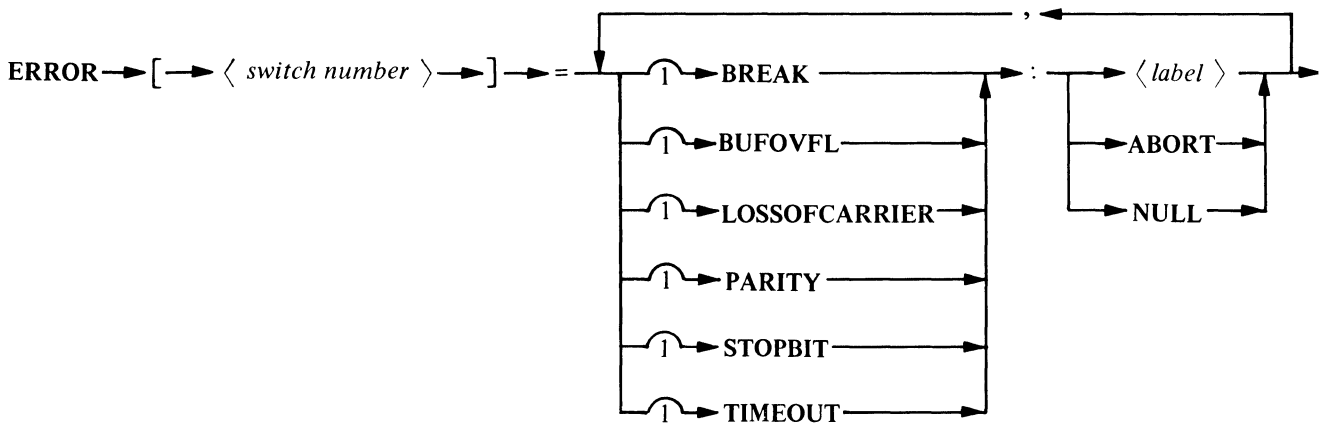
The ⟨*delay statement*⟩ provides a means to delay a specified period of time before control proceeds to the next statement. The ⟨*request definition*⟩ is suspended in a "sleep" state for the ⟨*delay time*⟩ specified.

Pragmatics

The "sleep" state induced by the ⟨*delay statement*⟩ allows the DCP to service other logical lines.

ERROR SWITCH STATEMENT

Syntax



Examples

```
ERROR [0] = BREAK: 0, BUFOVFL: NULL, LOSSOFCARRIER: ABORT, PARITY: 999,
            STOPBIT: 999. TIMEOUT: NULL.
ERROR [1] = BREAK: NULL.
ERROR [99] = BUFOVFL: NULL.
```

Semantics

The ⟨*error switch statement*⟩ is a non-executable statement that allows the programmer to define a set of default actions that are to be taken in a ⟨*receive statement*⟩ if the specified errors occur. ⟨*switch number*⟩ is an integer within a range of 0 to 8,388,607. The semantics of each option is described subsequently.

BREAK

The **BREAK** option variations cause the following actions if a break, that is, at least two character-times of a spacing line condition, is detected by the adapter cluster while receiving:

BREAK: NULL	sets **TRUE** the ⟨*bit variable*⟩ **BREAK [RECEIVE]**. Execution proceeds as if the break did not occur.
BREAK: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **BREAK [RECEIVE]**, and branches control to ⟨*label*⟩.
BREAK: ABORT	sets **TRUE** the ⟨*bit variable*⟩ **BREAK [RECEIVE]**, and executes an implicit **TERMINATE ERROR.**

BUFOVFL

The **BUFOVFL** option variations cause the following actions if the DCP is unable to service a Cluster Attention Needed (CAN) interrupt before the adapter cluster receives another character (thus destroying the previous character):

BUFOVFL: NULL	sets **TRUE** the ⟨*bit variable*⟩ **BUFOVFL**. Execution proceeds as if the error conditions did not occur.
BUFOVFL: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **BUFOVFL**, and branches control to ⟨*label*⟩.

Error Switch Statement – Continued

BUFOVFL: ABORT	sets **TRUE** the ⟨*bit variable*⟩ **BUFOVFL,** and executes an implicit **TERMINATE ERROR.**

LOSSOFCARRIER

The **LOSSOFCARRIER** option variations cause the following actions if a loss of carrier is detected while receiving.

LOSSOFCARRIER: NULL	sets **TRUE** the ⟨*bit variable*⟩ **LOSSOFCARRIER**. Execution proceeds as if the error did not occur.
LOSSOFCARRIER: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **LOSSOFCARRIER,** and branches control to ⟨*label*⟩ .
LOSSOFCARRIER: ABORT	sets **TRUE** the ⟨*bit variable*⟩ **LOSSOFCARRIER,** and executes an implicit **TERMINATE ERROR.**

There is one exception to the actions described in the above. If a loss of carrier is detected while receiving, and if the terminal is modem-connect, and if the terminal's ⟨*station definition*⟩ references a ⟨*modem definition*⟩ that contains the statement **LOSSOFCARRIER=DISCONNECT,** then an implicit disconnect is done, regardless of the action associated with **LOSSOFCARRIER** in the ⟨*error action statement*⟩ .

PARITY

The PARITY option variations cause the following actions if a parity bit error is detected by the adapter cluster:

PARITY: NULL	sets **TRUE** the ⟨*bit variable*⟩ **PARITY**. Execution proceeds as if the error did not occur.
PARITY: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **PARITY,** and branches control to ⟨*label*⟩ .
PARITY: ABORT	sets **TRUE** the ⟨*bit variable*⟩ **PARITY,** and executes a **TERMINATE ERROR.**

STOPBIT

The **STOPBIT** option variations cause the following actions if a stop bit error is detected by the adapter cluster:

STOPBIT: NULL	sets **TRUE** the ⟨*bit variable*⟩ **STOPBIT**. Execution proceeds as if the error did not occur.
STOPBIT: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **STOPBIT,** and branches control to ⟨*label*⟩ .
STOPBIT: ABORT	sets **TRUE** the ⟨*bit variable*⟩ **STOPBIT,** and executes a **TERMINATE ERROR.**

TIMEOUT

The **TIMEOUT** option variations of the **TIMEOUT** syntax shown below cause the actions described if the time required to receive a character exceeds the ⟨*timeout time*⟩. The ⟨*timeout time*⟩ is defined in the ⟨*terminal timeout statement*⟩, but can be overridden by including the (⟨*timeout time*⟩) or **(NULL)** syntax options in the ⟨*receive statement*⟩.

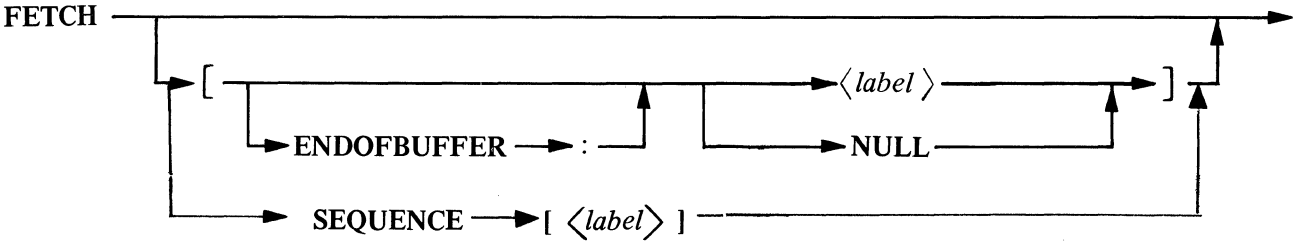
TIMEOUT: NULL	sets **TRUE** the ⟨*bit variable*⟩ **TIMEOUT**. Execution proceeds as if the error did not occur.
TIMEOUT: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **TIMEOUT,** and branches control to ⟨*label*⟩.
TIMEOUT: ABORT	sets **TRUE** the ⟨*bit variable*⟩ **TIMEOUT**, and executes a **TERMINATE ERROR.**

Pragmatics

An ⟨*error switch statement*⟩ must be associated with a ⟨*receive statement*⟩ by means of a ⟨*switch number*⟩ reference before any of the default actions are invoked. The ⟨*error switch statement*⟩ can appear in a ⟨*request definition*⟩ as many times as the programmer deems convenient, providing the following restriction is adhered to: within a given ⟨*request definition*⟩, ⟨*error switch statement*⟩*s* must have a unique ⟨*switch number*⟩, and all ⟨*error switch statement*⟩*s* must precede all executable statements in the procedure.

FETCH STATEMENT

Syntax



Examples

```
FETCH.
FETCH [10].
FETCH [ENDOFBUFFER:NULL].
```

Semantics

The execution of the ⟨*fetch statement*⟩ loads into **CHARACTER**, the character pointed to by the message text pointer and updates the pointer to point forward one character position.

When using the ⟨*fetch statement*⟩, provision should be made for taking action if the end-of-the-text buffer is encountered. The programmer can specify this action by including the optional syntax shown in the syntax diagram.

NULL specifies that no action should be taken.

⟨*label*⟩ specifies that control should branch to ⟨*label*⟩ if the end of buffer is encountered.

If the end of buffer is encountered and no action is specified, an implicit **TERMINATE ERROR** is executed.

For program documentation, the **ENDOFBUFFER** syntax can be added to the error action specification.

Supplementary Example

```
    INITIATE TRANSMIT.
3:  FETCH [ENDOFBUFFER:5].
    TRANSMIT CHAR.
    GO TO 3.
5:  FINISH TRANSMIT.
```

The **FETCH SEQUENCE** statement causes the individual characters which comprise the current sequence number to be loaded into **CHARACTER**. The digits are selected from the most significant to least significant digit. If no more digits remain or if sequence mode is not enabled, a branch to ⟨*label*⟩ is performed.

Thus, the construct:

TRANSMIT SEQUENCE

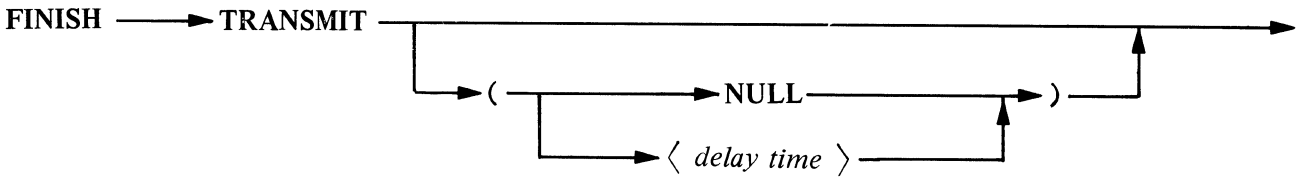
is equivalent to:

```
                INITIALIZE SEQUENCE.
GETSEQ:         FETCH SEQUENCE [ENDOFSEQ].
                TRANSMIT CHARACTER.
                GO TO GETSEQ.
ENDOFSEQ:
```

Prior to execution of this statement, the **DCP** sequence number logic must be initialized via the Initialize Sequence Statement.

FINISH STATEMENT

Syntax



Examples

```
FINISH TRANSMIT.
FINISH TRANSMIT (NULL).
FINISH TRANSMIT (3 SEC).
```

Semantics

The purpose of the ⟨*finish statement*⟩ is to complete the transmission of the last transmitted byte in preparation for receiving information. If the ⟨*delay time*⟩ is not specified or is NULL, the adapter cluster will wait for the completion of the transmission of the last byte and then wait an additional 2 milliseconds before allowing the DCP to proceed to the next statement. In addition, the line is placed in **RECEIVE** mode, although the adapter cluster will not sense any character for an additional 25 milliseconds. An **INITIATE RECEIVE** statement should precede any subsequent ⟨*receive statement*⟩ to override the 25 millisecond delay. If the ⟨*delay time*⟩ is specified and is non-null, the DCP will simply perform a delay for the specified time. An **INITIATE RECEIVE** statement is also required in this case to place the line in **RECEIVE** mode.

For example, the statement

```
FINISH TRANSMIT (3 SEC).
```

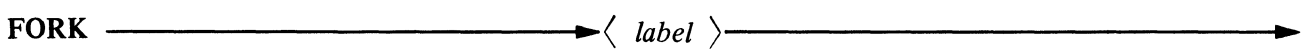
is equivalent to

```
FINISH TRANSMIT.
DELAY (3 SEC).
```

The **FINISH TRANSMIT (NULL)** form is equivalent to **FINISH TRANSMIT.**

FORK STATEMENT

Syntax



Example

```
FORK 10.
```

Semantics

The ⟨*fork statement*⟩ may be executed on either a primary or auxiliary line. It causes the other line to execute the code beginning at the label specified, while the first line continues executing in parallel.

If the other line had already been busy, it is not forked and the statement is a no-op; otherwise, being forked, it is automatically marked busy. The "busy" status is the same as that which NDL statements may reference **(LINE(BUSY), AUX LINE(BUSY))** so the NDL code running for one line or another may reset its busy status, so as to, even though running, allow itself to be forked to handle a more important function.

Pragmatics

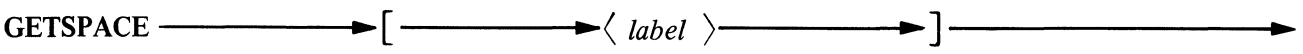
Synchronization problems can occur between the primary and auxiliary lines as a result of the ⟨*fork statement*⟩ executing the implicit **PAUSE**. The implicit **PAUSE** yields use of the DCP, to allow processing to to proceed on other lines. Thus, processing on the co-line is actually started before the **FORK**ing line exits the ⟨*fork statement*⟩. As a result, the programmer must, by some means (e.g., by setting and testing line **TOG**s), effect the synchronization of the lines. This is especially critical if the code contains ⟨*wait statement*⟩*s* and ⟨*continue statement*⟩*s*. The following example illustrates how full duplex lines could "hang" as a result of poor synchronization.

```
        FORK 10.
        WAIT.
          .
          .
          .
10:     CONTINUE.
        WAIT.
          .
          .
          .
```

Assume that the primary line executes the **FORK 10.** At that point, the primary line temporarily yields use of the DCP to other lines. The auxiliary line starts up and executes the **CONTINUE**. Since primary control is still at the ⟨*fork statement*⟩ and is not in a ⟨*wait statement*⟩, the auxiliary line **CONTINUE** acts as a no-op. Next, the auxiliary line executes the **WAIT**. When the primary line gets use of the processor again, it executes its **WAIT**. At this point, the primary and auxiliary lines are "hung", each **WAIT**ing for a **CONTINUE** from its co-line.

GETSPACE STATEMENT

Syntax



Example

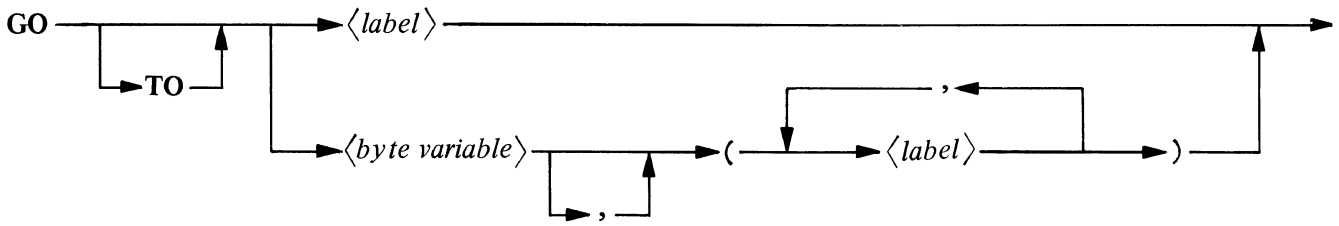
```
GETSPACE [10]
```

Semantics

The ⟨*getspace statement*⟩ provides the means for a Receive Request to explicitly acquire a message space for input. The message space (if obtained) is linked into the head of the Station Queue, thereby setting **STATION (QUEUED)** to **TRUE**. If there is no message space available at the time the ⟨*getspace statement*⟩ is executed, control branches to the ⟨*label*⟩. If a message space has already been acquired, this instruction acts as a no-op. This statement is also treated as a no-op if it appears in a Transmit Request.

GO TO STATEMENT

Syntax



Examples

```
GO GETNEXTCHAR
GO 10.
GO TO 10.
GO TO TOGS, (0,1,2,3).
GO TO STATION (5,9,12).
```

Semantics

The ⟨*go to statement*⟩ alters the path of control, that is, the sequential flow of statement execution, within a ⟨*request definition*⟩.

GO TO ⟨*label*⟩

This form of the ⟨*go to statement*⟩ unconditionally transfers control to the ⟨*label*⟩ specified.

GO TO ⟨*byte variable*⟩ . . .

This form of the ⟨*go to statement*⟩ provides a convenient means of dynamically selecting one or more ⟨*label*⟩*s* to which control could branch. The ⟨*label*⟩ to branch to is selected by using the ⟨*byte variable*⟩ as an index value. If N represents the number of ⟨*label*⟩*s* in the ⟨*go to statement*⟩, then the ⟨*label*⟩*s* are numbered 0 to N-1. The ⟨*label*⟩ corresponding to the index value is the ⟨*label*⟩ to which control branches. If the index value is greater than N-1, then control continues at the statement following the ⟨*go to statement*⟩.

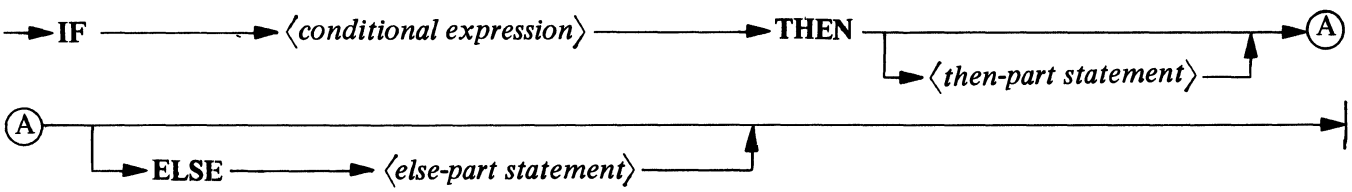
Supplementary Example

```
GO TO STATION (5,9,12).
% EXECUTION CONTINUES HERE IF STATION  > 2.
      .
      .
      .
5:  TOG [0]    = TRUE.
      .
      .
      .
9:  TOG [1]    = TRUE.
      .
      .
      .
12: TOG [2]    = TRUE.
      .
      .
      .
```

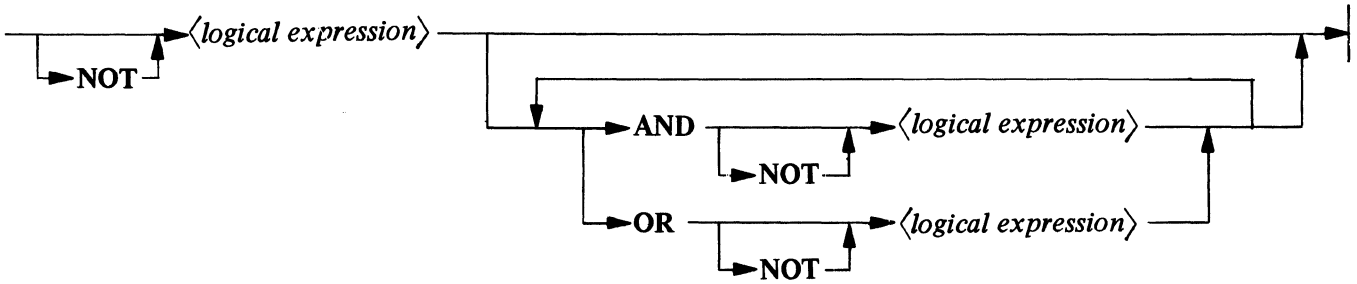
This example illustrates the **GO TO** ⟨*byte variable*⟩ construct of the ⟨*go to statement*⟩. The value of **STATION** determines the next statement to be executed. If the value of **STATION** is 0, control branches to the ⟨*label*⟩ 5; if the value of **STATION** is 1, control branches to ⟨*label*⟩ 9; and if the value of **STATION** is 2, control branches to ⟨*label*⟩ 12. If the value of **STATION** is greater than 2, control continues at the next sequential statement.

IF STATEMENT

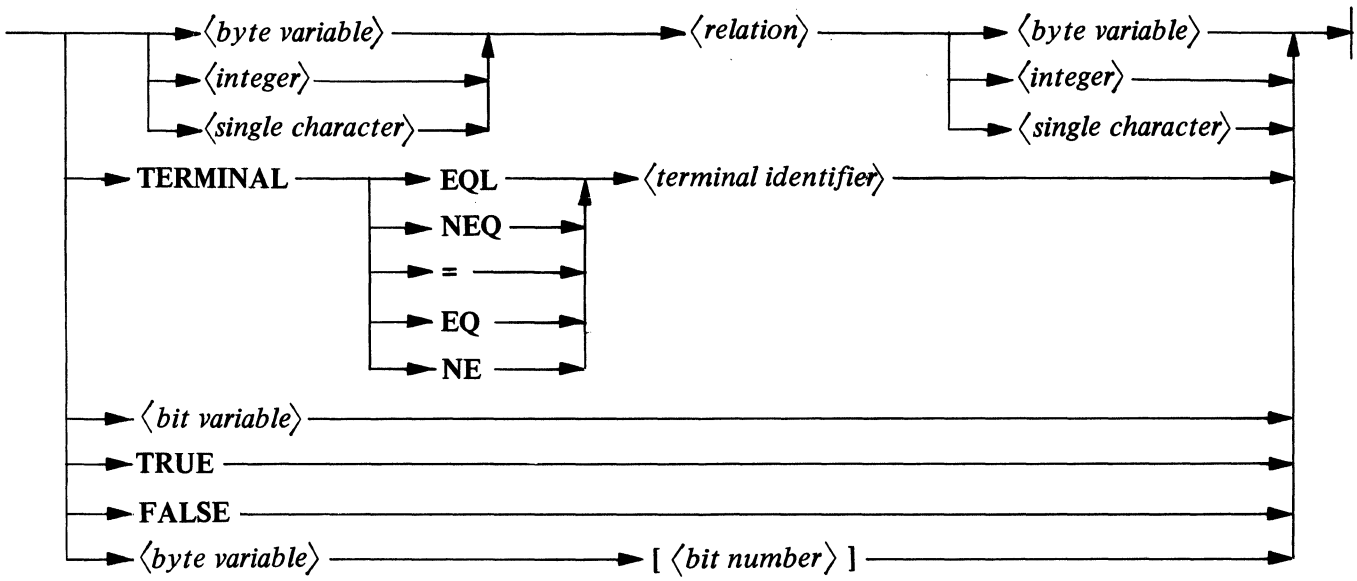
Syntax



⟨conditional expression⟩



⟨logical expression⟩



Examples

```
IF TRUE THEN.
IF TOG [0] THEN    TOG [0] = FALSE.
IF TALLY [0]  LSS TALLY [1]  THEN TALLY [0] = TALLY [1].
IF TERMINAL = TELETYPE THEN DELAY (150 MILLI).
IF TIMEOUT OR PARITY AND
     CHAR = 4"7F" THEN
GO TO ENDOFDATA.
IF CHARACTER = 4"FF" THEN
     INITIATE BREAK
ELSE
     BEGIN
     CHAR = 4"00".
     GO TO 0.
     END.
```

Semantics

The ⟨*if statement*⟩ causes a condition (i.e., a Boolean expression) to be evaluated. The subsequent path of program control depends on whether the condition is evaluated as **TRUE** or **FALSE**.

If the condition is **TRUE**, the ⟨*then-part statement*⟩ following the **THEN**, if present, is executed. Program control then resumes at the statement that follows the ⟨*if statement*⟩.

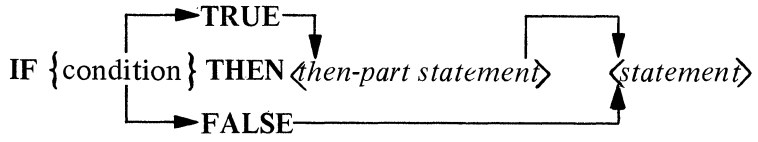
If the condition is **FALSE**, the ⟨*else-part statement*⟩ following the **ELSE** is executed or, if the **ELSE** ⟨*else-part statement*⟩ is omitted, program control resumes at the ⟨*statement*⟩ following the ⟨*if statement*⟩.

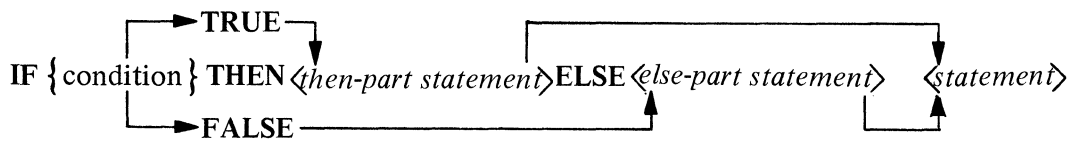
⟨*terminal identifier*⟩ has the syntactic form of ⟨*identifier*⟩ and is the name used in a subsequent **TERMINAL** definition which describes the physical attributes of a particular terminal type.

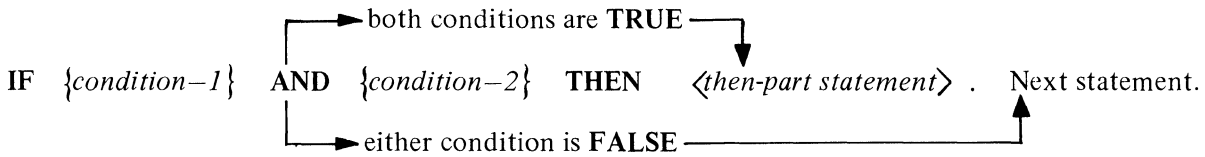
Note that no parentheses are allowed in the conditional expression, and that the normal precedence of "**NOT**" over "**AND**" over "**OR**" applies. The code generated for testing the condition is such that only those logical expressions needed to establish the truth or falsity of the condition are evaluated. The only time where evaluation of a logical expression (or the lack thereof) should matter is when **RECEIVE ADDRESS** is used as a byte variable.

The meanings of the relational operators are contained in table 5-6.

The following diagrams illustrate the above semantics.







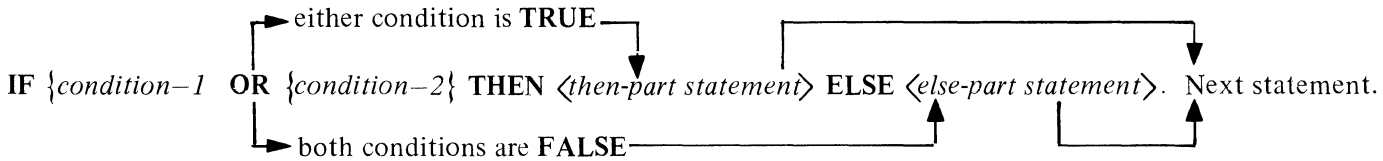
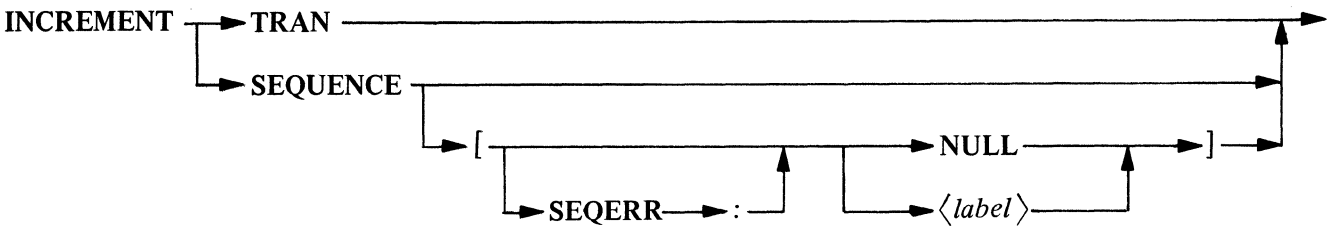


Table 5-6. Relational Operators

RELATIONAL OPERATOR	MEANING	SYNONYMS
LSS	Less than	< and LS
LEQ	Less than or equal to	LE
EQL	Equal to	= and EQ
NEQ	Not equal to	NE
GEQ	Greater than or equal to	GE
GTR	Greater than	> and GT

INCREMENT STATEMENT

Syntax



Examples

```
INCREMENT TRAN.
INCREMENT SEQUENCE [SEQERR:10].
INCREMENT SEQUENCE [NULL].
```

Semantics

INCREMENT TRAN

This construct of the ⟨*increment statement*⟩ is only allowed in those ⟨*request definition*⟩*s* in the ⟨*terminal request statement*⟩*s* of ⟨*terminal definition*⟩*s* that contain a ⟨*terminal transmission number length statement*⟩ defining the transmission number length as nonzero and non-**NULL**.

INCREMENT TRAN causes 1 to be added to the receive transmission number stored in the Station Table when it is executed in a Receive Request, and causes 1 to be added to the transmit transmission number stored in the Station Table when it is executed in a Transmit Request.

The transmission numbers are stored and incremented in EBCDIC.

If **INCREMENT TRAN** causes the transmission number to exceed (overflow) the size of the transmission number field, the carry is truncated and the result will be zeros (i.e., EBCDIC zeros) in that field.

INCREMENT SEQUENCE

This construct causes the sequence number stored in the DCP Station Table to be increased by the value of the increment (also stored in the DCP Station Table), providing that the station is in "sequence mode"; otherwise, this statement is a no-op.

When using the **INCREMENT SEQUENCE** construct, provision should be made for taking action if the increment caused the sequence number to exceed (overflow) the size of the sequence number field. The programmer can take such action by including the optional syntax. Failure to include overflow action results in an implicit **TERMINATE ERROR** if an overflow occurs.

SEQERR:NULL and **NULL** are semantically equivalent. These options set the **SEQERR** ⟨*bit variable*⟩ **TRUE**, and control continues at the next sequential instruction.

SEQERR:⟨*label*⟩ and ⟨*label*⟩are semantically equivalent. They cause the **SEQERR** ⟨*bit variable*⟩ to be set **TRUE**, and control to branch to ⟨*label*⟩.

Regardless of whether error action is specified or not, an overflow of the sequence number field destroys the contents of that field.

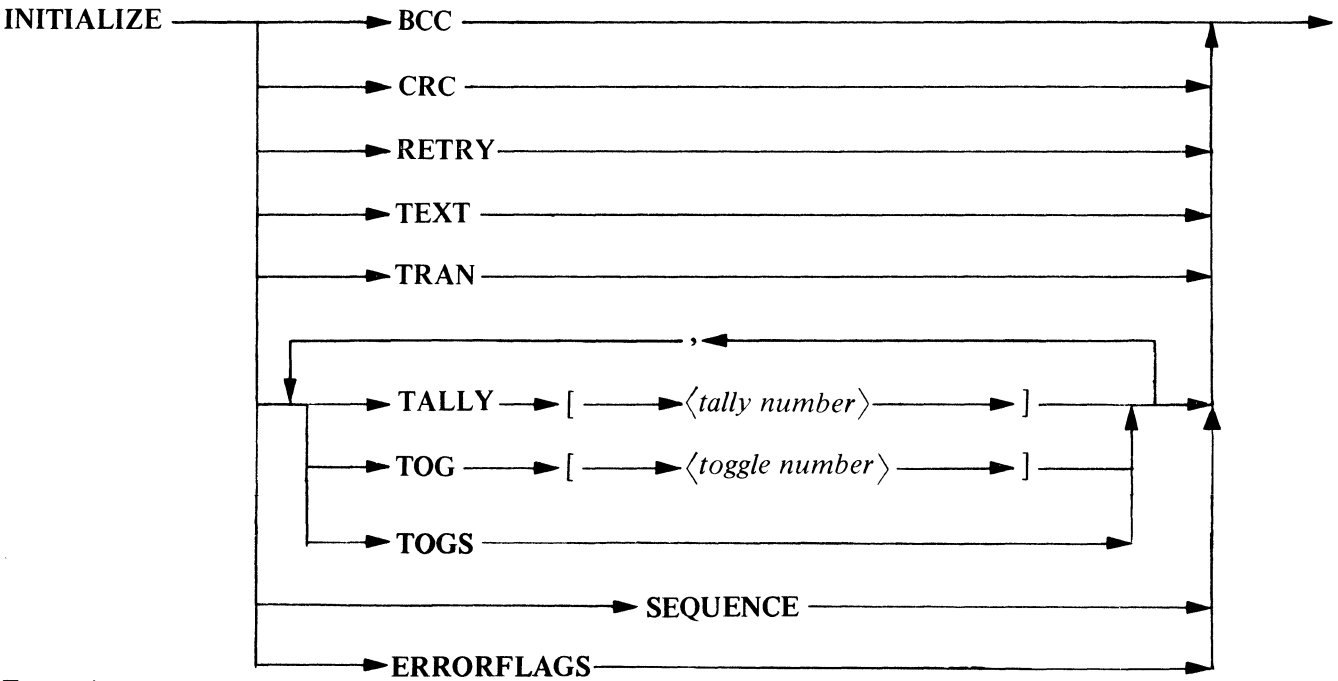
Pragmatics

A station is considered to be in sequence mode whenever its **SEQUENCE** ⟨*bit variable*⟩ is **TRUE.** **SEQUENCE** can be set **TRUE** only as a result of the Message Control System (MCS) executing the SET/RESET SEQUENCE MODE (TYPE = 49) DCWRITE. In addition, the TYPE 49 DCWRITE also stores the starting sequence number and increment in the appropriate fields of the DCP Station Table.

Sequence mode can be used for any application that the NDL programmer may see fit. Its use, however, requires common conventions between the NDL programmer and the MCS programmer. Burroughs has utilized sequence mode constructs in two ⟨*request definition*⟩*s* of SYMBOL/SOURCENDL: READTELETYPE and WRITETELETYPE. Both require the cooperation of SYSTEM/CANDE to effect the execution of those statements. The reader is referred to those ⟨*request definition*⟩*s* as an example of a particular application that Burroughs has implemented.

INITIALIZE STATEMENT

Syntax



Examples

```
INITIALIZE BCC.
INITIALIZE CRC.
INITIALIZE RETRY.
INITIALIZE TOGS.
INITIALIZE ERRORFLAGS.
```

Semantics

INITIALIZE BCC

This construct causes the ⟨*byte variable*⟩ **BCC** to be initialized for purposes of accumulating a Block Check Character. The value to which **BCC** is initialized is dependent upon the horizontal parity defined for the station's associated ⟨*terminal definition*⟩ (in the ⟨*terminal definition parity statement*⟩). If horizontal parity is defined as **HORIZONTAL:ODD**, then **BCC** is initialized to all ones (i.e., 4"FF"). If defined as **HORIZONTAL:EVEN**, then **INITIALIZE BCC** initializes **BCC** to all zeros (i.e., 4"00").

INITIALIZE CRC

This instruction initializes **CRC** to the initial value required for calculating the Cyclic Redundancy Check. Any ⟨*terminal definition*⟩ referencing a ⟨*request definition*⟩ (in the ⟨*terminal request statement*⟩) that contains this instruction must define the horizontal parity (in the ⟨*terminal parity statement*⟩) as **HORIZONTAL:CRC(16)**; otherwise a syntax error is generated.

INITIALIZE RETRY

This instruction causes the value stored in DCP INITIALRETRY to be stored DCP RETRY.

INITIALIZE TEXT

The function of this form is to initialize the message text pointer to zero. When initialized to zero, the message text pointer points to the first text character of the message.

INITIALIZE TRAN

This form causes zeroes (i.e., EBCDIC zeroes, 4"F0F0F0") to be stored in the appropriate Transmission Number fields of the Station Table. In a Receive Request, zeroes are stored in the Receive Transmission Number field; in a Transmit Request, zeroes are stored in the Transmit Transmission Number field.

INITIALIZE TALLY [⟨*tally number*⟩]

This form causes the specified station **TALLY** to be initialized from the appropriate message header field if a message is present; otherwise the specified **TALLY** is initialized to zero.

INITIALIZE TOG [⟨*toggle number*⟩]

This form causes the specified station **TOGGLE** to be initialized from the appropriate message field if a message is present; otherwise the specified **TOGGLE** is initialized **FALSE.**

NOTE

Only **TALLY [0] - TALLY [2]**, and **TOG [0] - TOG [7]**, can be **INITIALIZED** from an MCS message **HEADER**, or **STORE**d there.

INITIALIZE TOGS

This form initializes **TOGS [0 - 7]** from the current message.

INITIALIZE SEQUENCE

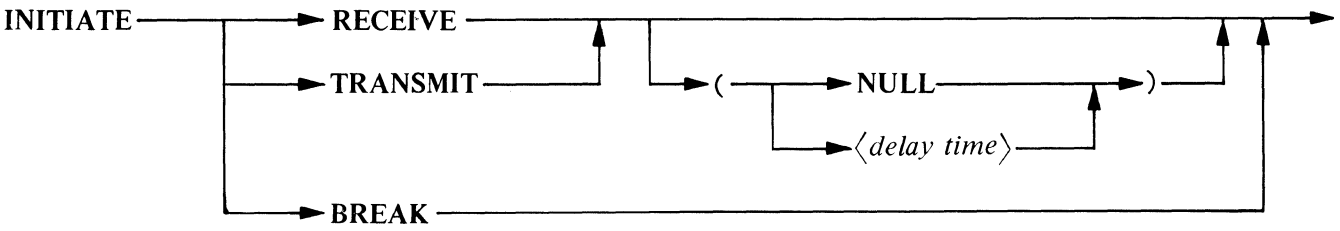
This form initializes the **DCP SEQUENCE** number logic. This must be done prior to using the **FETCH SEQUENCE** construct.

INITIALIZE ERRORFLAGS

This form clears the **ERROR FLAG** field in the Station Results word in the current message and the station table. This form also resets the **LAST ERROR FLAG** field.

INITIATE STATEMENT

Syntax



Examples

```
INITIATE RECEIVE.
INITIATE TRANSMIT (3 SEC).
INITIATE BREAK.
```

Semantics

INITIATE RECEIVE

The **INITIATE RECEIVE** construct causes the adapter cluster to initiate a receive delay calculated for the station. After the delay, the hardware is ready to receive information.

The amount of time delayed, referred to as the Initiate Receive delay, is unique to each station and is calculated at compile-time for each station. The algorithm that the compiler uses to calculate the Initiate Receive delay is described in the following three paragraphs.

a. If the ⟨*modem definition*⟩ referenced in the ⟨*station definition*⟩ (in the ⟨*station modem statement*⟩ defines the modem **NOISEDELAY** as being greater than zero, then the Initiate Receive delay is 2 milliseconds less than the combined ⟨*time*⟩s defined in the ⟨*modem noisedelay statement*⟩ and the ⟨*modem transmitdelay statement*⟩

b. If the modem **NOISEDELAY** is defined as zero and the modem **TRANSMITDELAY** is defined as being less than 7 milliseconds, then the Initiate Receive delay is zero.

c. If the modem **NOISEDELAY** is defined as zero and the modem **TRANSMITDELAY** is defined as being equal to or greater than 7 milliseconds, then the Initiate Receive delay is the lesser of 15 milliseconds or

$$(1.5 \text{ milliseconds} + \frac{\text{modem TRANSMITDELAY}}{2}).$$

The Initiate Receive delays for all stations assigned to a given line are maximized; the DCP stores this maximized value and uses it for the entire line.

The **NULL** option or the ⟨*delay time*⟩ option can be used to override the calculated Initiate Receive delay. **NULL** immediately readies the hardware so that it can receive information. ⟨*delay time*⟩ specifies a ⟨*time*⟩ to be used in place of the Initiate Receive delay.

Pragmatics

An **INITIATE RECEIVE** instruction should precede the first ⟨*receive statement*⟩ following a transmission. If it does not, there is a possibility that execution of the ⟨*receive statement*⟩ will be delayed for a period of time of up to 25 milliseconds. The cause of the 25-millisecond delay is described under the semantics of the ⟨*finish statement*⟩

INITIATE TRANSMIT

The **INITIATE TRANSMIT** construct causes the adapter cluster to be put in a transmit state after a calculated delay. The amount of time delayed is referred to as the Initiate Transmit Delay, and is unique to each station. It is derived by taking the greater of the **NOISEDELAY** ⟨*time*⟩ specified for the modem configured at the system end, or the **TURNAROUND** ⟨*time*⟩ specified by the station's ⟨*terminal definition*⟩.

The Initiate Transmit delays for all stations assigned to a given line are maximized; the DCP stores this maximized value and uses it for the entire line.

This construct must be executed prior to any attempt to **TRANSMIT**.

The **NULL** option or the ⟨*delay time*⟩ option can be used to override the calculated Initiate Transmit delay. **NULL** causes the adapter cluster to be put in a transmit state immediately. ⟨*delay time*⟩ specifies a ⟨*time*⟩ to be used in place of the Initiate Transmit delay.

INITIATE BREAK

The **INITIATE BREAK** construct causes binary zeroes to be transmitted on the line, thus changing the state of the line to a "spacing" condition. The line remains in the spacing condition until some subsequent instruction causes the adapter cluster to change the state of the line. Constructs that would change the line's state are **INITIATE TRANSMIT, INITIATE RECEIVE, FINISH TRANSMIT, BREAK** and **IDLE**.

PAUSE STATEMENT.

Syntax

Semantics

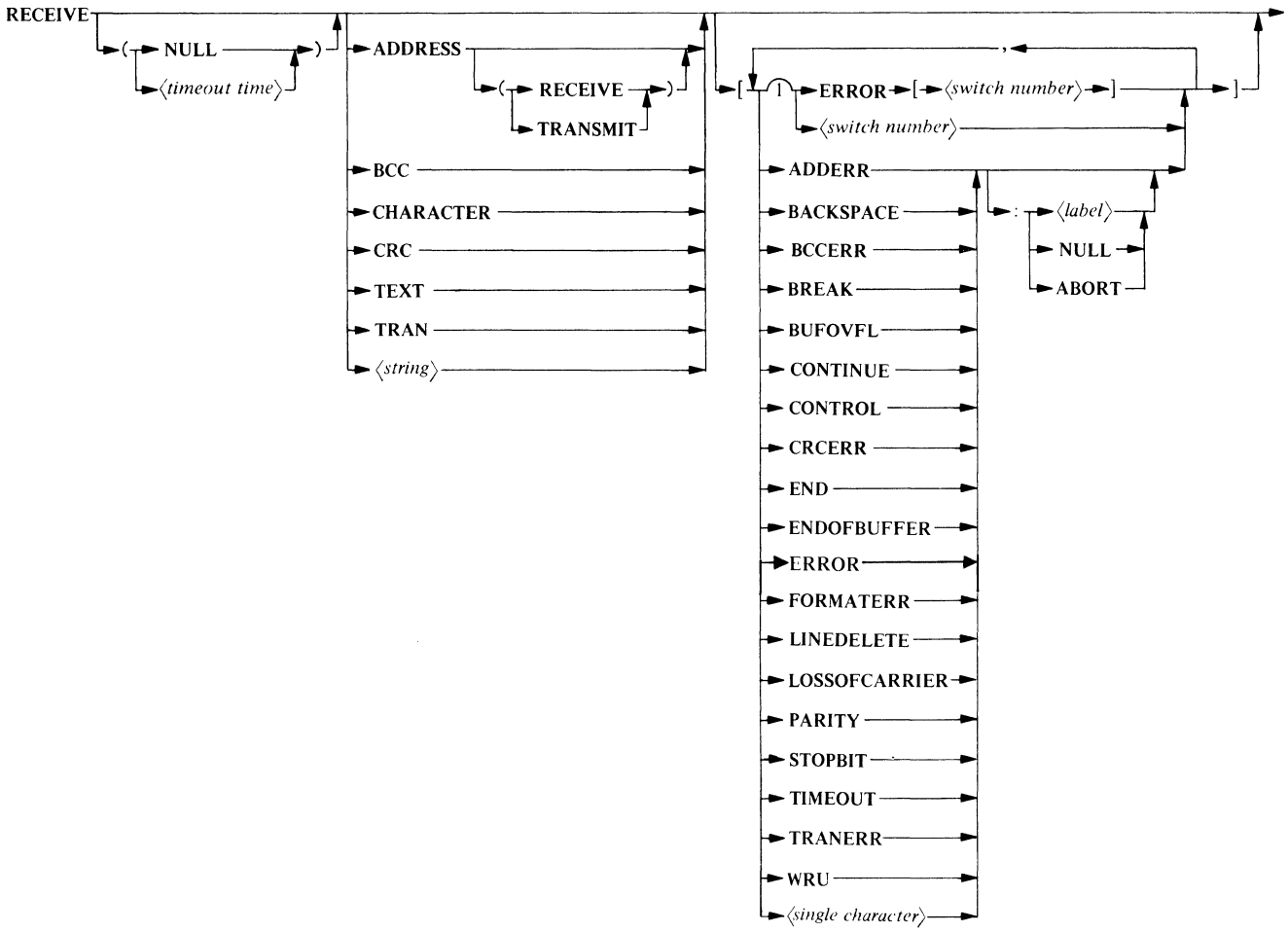
The ⟨*pause statement*⟩ suspends the ⟨*request definition*⟩ in a "sleep" state for a minimum period of time to allow the DCP to service other lines. It is recommended that a ⟨*pause statement*⟩ be used in any kind of loop that would tie up processor time and thereby prevent the servicing of other lines. The failure to do so results in a high number of timeout faults.

Pragmatics

Instances may occur in which the DCP requires an even greater period of "sleep" to service other lines. Repeated timeout faults, despite utilization of the ⟨*pause statement*⟩, are indications of such conditions. A greater period of "sleep" time can be effected by means of a ⟨*delay statement*⟩, with the ⟨*delay time*⟩ specified greater than "sleep" time effected by the ⟨*pause statement*⟩.

RECEIVE STATEMENT

Syntax



Examples

```
RECEIVE.
RECEIVE CHARACTER.
RECEIVE (3 SEC) ADDRESS (RECEIVE) [0, ADDERR:10].
RECEIVE (NULL)  [
                        PARITY:999,
                        LOSSOFCARRIER:999,
                        BACKSPACE:NULL,
                        END,
                        WRU:NULL
                        ].
RECEIVE CRC [ERROR [1], CRCERR:10].
RECEIVE "LITERAL STRING" [FORMATERR:NULL].
RECEIVE EOT SOH.
RECEIVE TEXT [END:10].
```

Semantics

The ⟨*receive statement*⟩ causes the adapter cluster to attempt to receive information from the appropriate logical line.

The following two syntax items define a maximum amount of time that the adapter cluster should wait for receipt of the first character, and then each subsequent character, if applicable, before assuming that the terminal has "timed out." If neither of these options is included, the ⟨*timeout time*⟩ defined (in the ⟨*terminal timeout statement*⟩) for the station's associated terminal type is implicitly used as the ⟨*timeout time*⟩ in this statement.

(NULL)

This option specifies that the adapter cluster should wait an infinite amount of time.

(⟨*timeout time*⟩ **)**

The ⟨*timeout time*⟩ defines a ⟨*time*⟩ that the adapter cluster should wait for a character. If this ⟨*time*⟩ is exceeded before receipt of a character, and the **TIMEOUT** syntax appears, then the action specified for **TIMEOUT** is taken (refer to **TIMEOUT**). If the ⟨*timeout time*⟩ is exceeded and **TIMEOUT** syntax does not appear, an implicit **TERMINATE ERROR** is executed.

The following syntax options define the nature of the information to be received, the amount of information to be received, and how the information is to be handled. If none of the options are used, it is semantically equivalent to specifying **CHARACTER** (e.g., "**RECEIVE.**" is semantically equivalent to "**RECEIVE CHARACTER.**").

ADDRESS

The proper number of address characters (as defined by the station's associated ⟨*terminal definition*⟩ in the ⟨*terminal address size statement*⟩) are received and checked for agreement against the actual address characters defined in the ⟨*station address statement*⟩. If the address characters do not correspond, an address error condition results; if the **ADDERR** syntax appears, then the specified action is taken. Otherwise an implicit **TERMINATE ERROR** is executed. (Refer to the **ADDERR** semantics.)

ADDRESS (RECEIVE)

This option is equivalent to **ADDRESS**, except that **ADDRESS (RECEIVE)** must be used when an address pair is defined in the ⟨*station address statement*⟩ and the programmer needs to check for the proper receive address.

ADDRESS (TRANSMIT)

This option is equivalent to **ADDRESS**, except that **ADDRESS (TRANSMIT)** must be used when an address pair is defined in the ⟨*station address statement*⟩ and the programmer needs to check for the proper transmit address.

BCC

One character is received and checked against the ⟨*byte variable*⟩ **BCC**. If the character received and **BCC** are not equal, a Block Check Character error condition results; if the **BCCERR** syntax appears, then the specified action is taken. Otherwise an implicit **TERMINATE ERROR** is executed.

Presumably, if the **RECEIVE BCC** construct appears, the programmer has defined horizontal parity in the ⟨*terminal parity statement*⟩, and the accumulated Block Check Character is contained in **BCC**.

CHARACTER

One character is received and stored in **CHARACTER.**

CRC

Two characters are received. The first character is checked against **CRC [0]**, and the second compared against **CRC [1]**. If the characters received and **CRC** are not equal, a Cyclic Redundancy Check error condition results; if the **CRCERR** syntax appears, then specified action is taken. Otherwise an implicit **TERMINATE ERROR** is executed.

Presumably, if the **RECEIVE CRC** instruction appears, the programmer has defined horizontal parity as **HORIZONTAL:CRC(16)** in the ⟨*terminal parity statement*⟩, and the Cyclic Redundancy Check is contained in **CRC [0]** and **CRC [1]**.

TEXT

Characters are received into **CHARACTER** and stored in the text portion of the message space obtained until either a syntax option results in a branch from the ⟨*receive statement*⟩, or a non-recoverable error, such as a disconnect, occurs. If the occurrence of a particular character results in a branch outside of the ⟨*receive statement*⟩ (as specified by a syntax option), then that character is not stored but remains in **CHARACTER.**

The **RECEIVE TEXT** construct is, in effect, the same as the following loop:

```
1: RECEIVE CHARACTER.
   STORE CHARACTER.
   GO TO 1.
```

In nearly all cases, the ⟨*receive statement*⟩ should contain optional syntax to avoid the "endless" loop and an eventual implicit **TERMINATE ERROR** as a result of a timeout, end-of-buffer condition, etc.

TRAN

The proper number of transmission number characters (as defined by the station's associated ⟨*terminal definition*⟩ in the ⟨*terminal transmission number length statement*⟩) are received and checked for agreement with the Receive Transmission Number maintained in the DCP Station Table. If the characters received and the Receive Transmission Number are not equal, a transmission number error results. If the **TRANERR** syntax appears, then specified action is taken; otherwise an implicit **TERMINATE ERROR** is executed.

⟨*string*⟩

The number of characters as indicated by the length of the ⟨*string*⟩ are received and checked against those characters in the ⟨*string*⟩. If the ⟨*string*⟩ and the characters received are not equal, then a format error condition results. If the **FORMATERR** syntax option appears, then that action is taken; otherwise an implicit **TERMINATE ERROR** is executed.

The following syntax options specify actions to be taken upon either the receipt of defined characters or occurrences of specific error conditions.

Receive Statement – Continued

ERROR [⟨*switch number*⟩]

This syntax option associates a previously defined Error Switch with the ⟨*receive statement*⟩. This allows the programmer to associate a set of previously defined error actions with the ⟨*receive statement*⟩, thus reducing the amount of coding required for each ⟨*receive statement*⟩. **BREAK**, **BUFOVFL**, **LOSSOFCARRIER**, **PARITY**, **STOPBIT**, and **TIMEOUT** syntax options are not allowed if the **ERROR** [⟨*switch number*⟩] syntax appears in the ⟨*receive statement*⟩. Refer to the ⟨*error switch statement*⟩ for more information.

⟨*switch number*⟩

Semantically equivalent to **ERROR** [⟨*switch number*⟩].

ADDERR

The **ADDERR** option variations cause the following actions if an address error is detected when an attempt is made to receive a terminal's address characters:

ADDERR	sets **TRUE** the **ADDERR** ⟨*bit variable*⟩ and branches control to the next sequential statement.
ADDERR:NULL	sets **TRUE** the **ADDERR** ⟨*bit variable*⟩. Execution proceeds as if the error condition did not occur.
ADDERR:⟨*label*⟩	sets **TRUE** the **ADDERR** ⟨*bit variable*⟩ and branches control to ⟨*label*⟩.
ADDERR:ABORT	not allowed.

BACKSPACE

The following **BACKSPACE** option variations cause the following actions if the terminal's backspace character (as defined by the ⟨*terminal backspace character statement*⟩) is received:

BACKSPACE	moves the message text pointer backwards one character position, and branches control to the next sequential statement.
BACKSPACE:NULL	moves the message text pointer backwards one character. Control remains within the ⟨*receive statement*⟩ if of the form **RECEIVE TEXT**.
BACKSPACE: ⟨*label*⟩	moves the message text pointer backwards one character, and branches control to ⟨*label*⟩.
BACKSPACE:ABORT	not allowed.

BCCERR

The following **BCCERR** option variations cause the following actions if the character received is not equal to the data stored in BCC.

BCCERR	sets **TRUE** the ⟨*bit variable*⟩ **BCCERR**, and branches control to the next sequential statement.
BCCERR:NULL	sets **TRUE** the ⟨*bit variable*⟩ **BCCERR**. Execution proceeds as if the error condition did not occur.

BCCERR:⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **BCCERR** and branches control to ⟨*label*⟩.
BCCERR:ABORT	not allowed.

BREAK

The **BREAK** option variations cause the following actions as if a break, that is, at least two character-times of a spacing line condition, is detected by the adapter cluster while receiving:

BREAK	sets **TRUE** the ⟨*bit variable*⟩ **BREAK [RECEIVE]**, and branches control to the next sequential statement.
BREAK:NULL	sets **TRUE** the ⟨*bit variable*⟩ **BREAK [RECEIVE]**. Execution proceeds as if the break did not occur.
BREAK:⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **BREAK [RECEIVE]**, and branches control to ⟨*label*⟩.
BREAK:ABORT	sets **TRUE** the ⟨*bit variable*⟩ **BREAK [RECEIVE]**, and executes an implicit **TERMINATE ERROR**.

BUFOVFL

The following variations of the **BUFOVFL** option cause the following actions if the DCP is unable to service a Cluster Attention Needed (CAN) interrupt before the Adapter Cluster receives another character (thus destroying the previous character):

BUFOVFL	sets **TRUE** the ⟨*bit variable*⟩ **BUFOVFL**, and branches control to the next sequential statement.
BUFOVFL:NULL	sets **TRUE** the ⟨*bit variable*⟩ **BUFOVFL**. Execution proceeds as if the error condition did not occur.
BUFOVFL:⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **BUFOVFL**, and branches control to ⟨*label*⟩.
BUFOVFL:ABORT	sets **TRUE** the ⟨*bit variable*⟩ **BUFOVFL**, and executes an implicit **TERMINATE ERROR**.

CONTINUE

This item is allowed only in ⟨*receive statement*⟩*s* of ⟨*control definition*⟩*s* and ⟨*request definition*⟩*s* that are written to communicate with full duplex terminal types. **CONTINUE** syntax causes action as described below if the co-line executes a ⟨*continue statement*⟩ before all information specified by the ⟨*receive statement*⟩ is received.

CONTINUE	branches control to the next sequential statement.
CONTINUE:NULL	causes no action. Execution proceeds as if the ⟨*continue statement*⟩ had not been executed.
CONTINUE: ⟨*label*⟩	branches control to ⟨*label*⟩.
CONTINUE:ABORT	not allowed.

CONTROL

The following variations of the **CONTROL** option cause the following actions if the control character of the station (as defined in the ⟨*station control character statement*⟩) is received:

CONTROL	sets TRUE the ⟨*bit variable*⟩ **CONTROLFLAG**, and branches control to the next sequential statement.
CONTROL:NULL	sets TRUE the ⟨*bit variable*⟩ **CONTROL FLAG**, and execution continues if the character was not the station's control character.
CONTROL: ⟨*label*⟩	sets TRUE the ⟨*bit variable*⟩ **CONTROLFLAG**, and branches control to ⟨*label*⟩.
CONTROL:ABORT	not allowed.

CRCERR

The following variations of the **CRCERR** option cause the following actions if the first character received does not equal **CRC [0]** , or the second character received does not equal **CRC [1]** .
(This item is appropriate only for the **RECEIVE CRC** construct of the ⟨*receive statement*⟩ ; refer to the CRC option.)

CRCERR	sets TRUE the ⟨*bit variable*⟩ **CRCERR**, and branches control to the next sequential statement.
CRCERR:NULL	sets **TRUE** the ⟨*bit variable*⟩ **CRCERR**. Execution proceeds as if the error condition did not occur.
CRCERR: ⟨*label*⟩	sets TRUE the ⟨*bit variable*⟩ **CRCERR**, and branches control to ⟨*label*⟩.
CRCERR:ABORT	not allowed.

END

The following variations of the **END** option cause the following actions if the "end" character of the station (as defined by the ⟨*terminal end character statement*⟩ in the ⟨*terminal definition*⟩ associated with the station) is received:

END	causes control to branch to the next sequential statement.
END:NULL	causes no action. Execution proceeds as if the character was not the "end" character.
END: ⟨*label*⟩	branches control to ⟨*label*⟩.
END:ABORT	not allowed.

ENDOFBUFFER

This syntax item is allowed in the **RECEIVE TEXT** contruct of the ⟨*receive statement*⟩. The variations of the **ENDOFBUFFER** option shown below cause the following actions if either of the following conditions arises:

a. There is no message space and an attempt is made to store information into a message space (the store function is an implicit action of the **RECEIVE TEXT** construct), or

b. The number of characters stored in the message exceeds the maximum allowed (the maximum is defined by either the ⟨*terminal maxinput statement*⟩ or the ⟨*terminal buffer size statement*⟩).

ENDOFBUFFER	sets **TRUE** the ⟨*bit variable*⟩ **ENDOFBUFFER**, and branches control to the next sequential statement.
ENDOFBUFFER:NULL	sets **TRUE** the ⟨*bit variable*⟩ **ENDOFBUFFER**. Execution proceeds as if the error did not occur.
ENDOFBUFFER: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **ENDOFBUFFER**, and branches control to ⟨*label*⟩.
ENDOFBUFFER:ABORT	not allowed.

ERROR

The **NDL** programmer may specify an ⟨*error lable*⟩ which is branched to when any error action specifiable in the ⟨*error switch statement*⟩ occurs. The Error Flag Field may then be interrogated to determine the error and the desired recovery steps may be performed.

⟨*error labels*⟩ may not be used with other ⟨*error label*⟩s, error switches, or with any error action specifiable in an ⟨*error switch statement*⟩.

ERROR	sets **TRUE** the appropriate ⟨*bit variable*⟩ (**TIMECUT**, **BUFOVFL**, **LOSSOFCARRIER**, **STOPBIT**, **PARITY** or **BREAK**), and branches control to the next sequential statement.
ERROR:NULL	sets **TRUE** the appropriate ⟨*bit variable*⟩. Execution proceeds as if no error occurred.
ERROR:⟨*label*⟩	sets **TRUE** the appropriate ⟨*bit variable*⟩, and branches control to the ⟨*label*⟩.
ERROR:ABORT	sets **TRUE** the appropriate ⟨*bit variable*⟩, and executes a **TERMINATE ERROR**.

FORMATERR

The following variations of the **FORMATERR** option cause the following actions if the characters received are not equal to those in the ⟨*string*⟩ (this item is appropriate only for the **RECEIVE** ⟨*string*⟩ construct of the ⟨*receive statement*⟩):

FORMATERR	sets **TRUE** the ⟨*bit variable*⟩ **FORMATERR**, and branches control to the next sequential statement.

FORMATERR:NULL	sets **TRUE** the ⟨*bit variable*⟩ **FORMATERR**. Execution proceeds as if the error did not occur.
FORMATERR: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **FORMATERR**, and branches control to ⟨*label*⟩.
FORMATERR:ABORT	not allowed.

LINEDELETE

The following variations of the **LINE DELETE** option cause the following actions if the station's linedelete character is received (the **LINEDELETE** character is defined by the ⟨*terminal linedelete character statement*⟩):

LINEDELETE	alters the value of the message text pointer to point to the first character position in the message text, and branches control to the next sequential statement.
LINEDELETE:NULL	alters the value of the message text pointer to point to the first character position in the message text, and execution proceeds as if the character was not the linedelete character.
LINEDELETE: ⟨*label*⟩	alters the value of the message text pointer to point to the first character position in the message text, and branches control to ⟨*label*⟩.
LINEDELETE:ABORT	not allowed.

LOSSOFCARRIER

The following variations of the **LOSSOFCARRIER** syntax cause the following actions if a loss of carrier is detected while receiving.

LOSSOFCARRIER	sets **TRUE** the ⟨*bit variable*⟩ **LOSSOFCARRIER**, and branches control to the next sequential statement.
LOSSOFCARRIER:NULL	sets **TRUE** the ⟨*bit variable*⟩ **LOSSOFCARRIER**. Execution proceeds as if the error did not occur.
LOSSOFCARRIER: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **LOSSOFCARRIER**, and branches control to ⟨*label*⟩.
LOSSOFCARRIER:ABORT	sets **TRUE** the ⟨*bit variable*⟩ **LOSSOFCARRIER**, and executes an implicit **TERMINATE ERROR.**

There is one exception to the actions described above. If a loss of carrier is detected while receiving, and if the terminal is modem-connect, and if the terminal's ⟨*station definition*⟩ references a ⟨*modem definition*⟩ that contains the construct **LOSSOFCARRIER=DISCONNECT**, then an implicit disconnect is done, regardless of the action specified.

PARITY

The following variations of the **PARITY** option cause the following actions if a parity bit error is detected by the adapter cluster:

PARITY	sets **TRUE** the ⟨*bit variable*⟩ **PARITY**, and branches control to the next sequential statement.
PARITY:NULL	sets **TRUE** the ⟨*bit variable*⟩ **PARITY**. Execution proceeds as if the error did not occur.
PARITY: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **PARITY**, and branches control to ⟨*label*⟩.
PARITY:ABORT	sets **TRUE** the ⟨*bit variable*⟩ **PARITY**, and executes a **TERMINATE ERROR**.

STOPBIT

The following variations of the **PARITY** option cause the described actions if a stop bit error is detected by the adapter cluster:

STOPBIT	sets **TRUE** the ⟨*bit variable*⟩ **STOPBIT**, and branches control to the next sequential statement.
STOPBIT:NULL	sets **TRUE** the ⟨*bit variable*⟩ **STOPBIT**. Execution proceeds as if the error did not occur.
STOPBIT: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **STOPBIT**, and branches control to ⟨*label*⟩.
STOPBIT:ABORT	sets **TRUE** the ⟨*bit variable*⟩ **STOPBIT**, and executes a **TERMINATE ERROR.**

TIMEOUT

The variations of the **TIMEOUT** syntax shown below cause the actions described if the time required to receive a character exceeds the ⟨*timeout time*⟩. The ⟨*timeout time*⟩ is defined in the ⟨*terminal timeout statement*⟩, but can be overridden by including the (⟨*timeout time*⟩) or **(NULL)** syntax options in the ⟨*receive statement*⟩.

TIMEOUT	sets **TRUE** the ⟨*bit variable*⟩ **TIMEOUT**, and branches control to the next sequential statement.
TIMEOUT:NULL	sets **TRUE** the ⟨*bit variable*⟩ **TIMEOUT**. Execution proceeds as if the error did not occur.
TIMEOUT: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **TIMEOUT**, and branches control to ⟨*label*⟩.
TIMEOUT:ABORT	sets **TRUE** the ⟨*bit variable*⟩ **TIMEOUT**, and executes a **TERMINATE ERROR.**

TRANERR

The following variations of the **TRANERR** option cause the described actions if the characters received and the Receive Transmission Number stored in the Station Table are not equal (this item is allowed only in the **RECEIVE TRAN** construct of the ⟨*receive statement*⟩):

TRANERR	sets **TRUE** the ⟨*bit variable*⟩ **TRANERR**, and branches control to the next sequential statement.
TRANERR:NULL	sets **TRUE** the ⟨*bit variable*⟩ **TRANERR**. Execution proceeds as if the error did not occur.
TRANERR: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **TRANERR**, and branches control to ⟨*label*⟩.
TRANERR:ABORT	not allowed.

WRU

The following variations of the **WRU** syntax cause the following actions if the WRU character of the station is received (the ⟨*station WRU character statement*⟩ defines the **WRU** character):

WRU	sets **TRUE** the **WRU** ⟨*bit variable*⟩, and branches control to the next sequential statement.
WRU:NULL	sets **TRUE** the **WRU** ⟨*bit variable*⟩, and execution proceeds as if the character received was not the **WRU** character.
WRU: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **WRU**, and branches control to ⟨*label*⟩.
WRU:ABORT	not allowed.

⟨*single character*⟩

The following variations of the ⟨*single character*⟩ syntax cause the following actions if a character received is equal to the *single character* :

⟨*single character*⟩	branches control to the next sequential statement.
⟨*single character*⟩**:NULL**	causes no action. Execution proceeds as if the character received was not equal to the ⟨*single character*⟩.
⟨*single character*⟩:⟨*label*⟩	branches control to ⟨*label*⟩.
⟨*single character*⟩**:ABORT**	not allowed.

The allowable combinations of the ⟨*receive statement*⟩ syntax options are defined in table 5–7 below. The **(NULL)** and (⟨*timeout time*⟩) options are allowed in any construct of the ⟨*receive statement*⟩. Allowed combinations of the other syntax options are denoted by a "X" in the appropriate columns and rows.

Table 5-7. Allowable Combinations for ⟨*receive statement*⟩

	ADDERR	**BACKSPACE**	**BCCERR**	**BREAK**	**BUFOVFL**	**CONTINUE**	**CONTROL**	**CRCERR**	**END**	**ENDOFBUFFER**	**FORMATERR**	**LINEDELETE**	**LOSSOFCARRIER**	**PARITY**	**STOPBIT**	**TIMEOUT**	**TRANERR**	**WRU**	⟨*single character*⟩
ADDRESS	X			X	X	X							X	X	X	X			
ADDRESS(RECEIVE)	X			X	X	X							X	X	X	X			
ADDRESS(TRANSMIT)	X			X	X	X							X	X	X	X			
BCC			X	X	X	X							X	X	X	X			
CHARACTER		X		X	X	X	X		X			X	X	X	X	X		X	X
CRC				X	X	X		X					X	X	X	X			
TEXT		X		X	X	X	X		X	X		X	X	X	X	X		X	X
TRAN				X	X	X							X	X	X	X	X		
⟨*string*⟩				X	X	X					X		X	X	X	X			

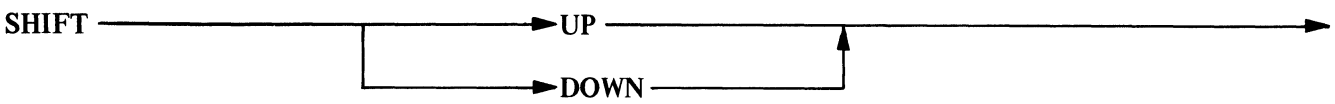
Supplementary Examples

Statement	Explanation
RECEIVE (3 SEC) [TIMEOUT:10].	Causes the adapter cluster to attempt to receive a character. If the character is not received within 3 seconds, the ⟨*bit variable*⟩ **TIMEOUT** is set **TRUE** and control branches to **10**.
RECEIVE ADDRESS [ADDERR:99].	If the character(s) received do not equal those defined in the ⟨*station address statement*⟩, the ⟨*bit variable*⟩ **ADDERR** is set **TRUE**, and control branches to **99**.
RECEIVE CHARACTER [CONTINUE:10, CONTROL:20, TIMEOUT:30, "*":40].	This statement would only be allowed in a ⟨*control definition*⟩ or ⟨*request definition*⟩ that is written to communicate with full duplex terminal types, because it contains the **CONTINUE** item. **CONTINUE:10** would cause a branch to 10 if the co-line ⟨*control definition*⟩ executes a ⟨*continue statement*⟩ before a character is received.

Statement	Explanation
	CONTROL:20 would set **CONTROLFLAG TRUE** and branch to **20** if the character received is the station's control character. **TIMEOUT:30** would set **TIMEOUT TRUE** and branch to 30 if a character is not received within the ⟨*timeout time*⟩ defined in the ⟨*terminal timeout statement*⟩. **"*":40** would cause a branch to **40** if the character received is the asterisk character.
RECEIVE [ERROR[0]].	An attempt is made to receive one character and store it in **CHARACTER**, If any errors described in the associated ⟨*error switch statement*⟩ occur while receiving, then the action defined in that ⟨*error switch statement*⟩ is taken.
RECEIVE [0].	An attempt is made to receive one character and store it in **CHARACTER**. If any errors described in the associated ⟨*error switch statement*⟩ occur while receiving, then the action defined in that ⟨*error switch statement*⟩ is taken.
RECEIVE (1 SEC) TEXT [LINEDELETE:NULL, CONTROL:NULL].	**LINEDELETE:NULL** causes the message text pointer to be set to the first character position if the linedelete character (as defined in the ⟨*terminal linedelete character statement*⟩ is received, and characters continue to be received and stored in the message text beginning at the first character position. **CONTROL:NULL** causes the ⟨*bit variable*⟩ **CONTROLFLAG** to be set **TRUE** if the control character of the station (defined in the ⟨*station control character statement*⟩) is received, and characters continue to be received.

SHIFT STATEMENT

Syntax



Semantics

The ⟨*shift statement*⟩ is to be used in a ⟨*control definition*⟩ that communicates with stations using the Baudot (5-bit) character code set. (The character code set is defined in the ⟨*terminal code statement*⟩ of the associated ⟨*terminal definition*⟩.)

SHIFT UP indicates that received characters are to be translated to their respective uppercase graphics (usually referred to as FIGS).

SHIFT DOWN indicates that received characters are to be translated to their respective lowercase graphics (usually referred to as LTRS).

If the station does not use Baudot code, the ⟨*shift statement*⟩ acts as a no-op.

Pragmatics

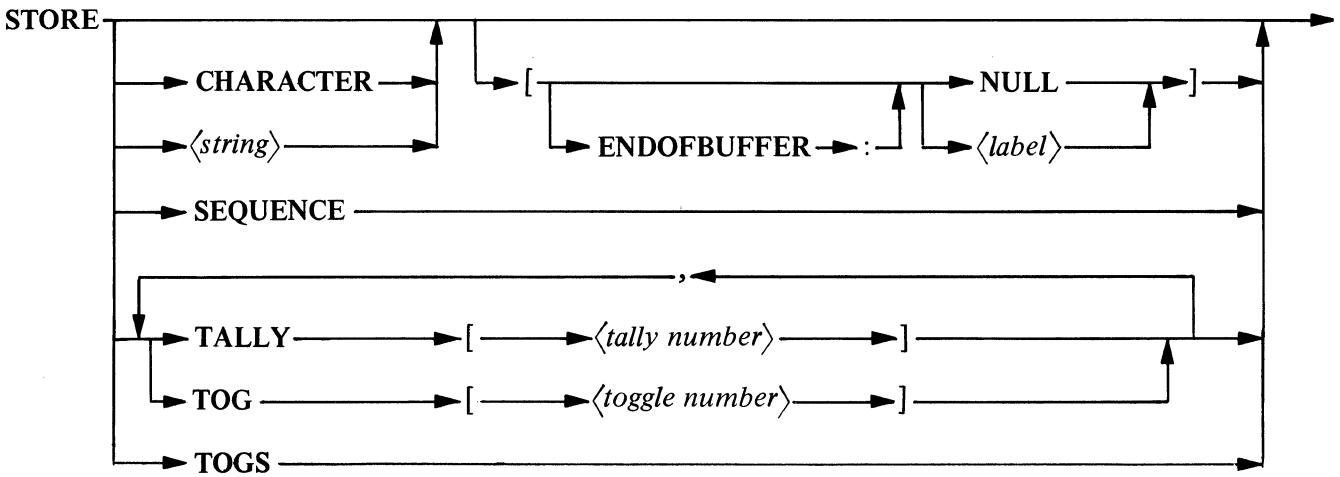
In the Baudot character code set, most bit patterns have two graphic representations; one is referred to as FIGS (the uppercase graphic), and the other as LTRS (the lowercase graphic).

When transmitting to a terminal that uses Baudot code, the terminal prints LTRS until it receives a specially designated character indicating that it should shift to printing FIGS. The terminal continues printing the FIGS until it receives a specially designated character indicating that it should resume printing the LTRS.

When information is received from a terminal that uses Baudot, the same conventions hold true; that is, the terminal communicates whether FIGS or LTRS follow, by the transmission of a designated character. The terminal initially transmits LTRS.

STORE STATEMENT

Syntax



Examples

```
STORE.
STORE CHARACTER [ENDOFBUFFER:20].
STORE "ABC" [NULL].
STORE SEQUENCE.
STORE TALLY [0].
STORE TOG [0], TOG [1], TALLY [0].
STORE TOGS.
```

Semantics

STORE

This form is semantically equivalent to the **STORE CHARACTER** construct.

STORE CHARACTER

This form causes the data contained in **CHARACTER** to be stored in the message space. If no message space is associated with the ⟨*request definition*⟩, then an implicit ⟨*getspace statement*⟩ is executed. The data is stored in the character position pointed to by the message text pointer, and the text pointer is updated after the **STORE** to point to the next forward character position.

It is possible to encounter the end-of-the-text buffer when using this instruction. It is recommended that the optional syntax be included whenever using this statement. The optional syntax specifies action to be taken if the end of buffer is encountered. The **NULL** option specifies that the only action that should be taken is to set **ENDOFBUFFER** to **TRUE**. The ⟨*label*⟩ option specifies that the only action that should be taken if the **ENDOFBUFFER** is encountered is that control should branch to ⟨*label*⟩ and also set **ENDOFBUFFER TRUE**. The **ENDOFBUFFER:** part can be included for documentation. An implicit **TERMINATE ERROR** is executed if no end-of-buffer action is specified.

STORE ⟨*string*⟩

This form causes ⟨*string*⟩ to be stored in the message space. If no message space is currently associated with the ⟨*request definition*⟩, an implicit ⟨*getspace instruction*⟩ is executed. The ⟨*string*⟩ is stored in the message space beginning at the character position pointed to by the message text pointer, and the text pointer is updated after the **STORE** to point to the first character position following the ⟨*string*⟩.

This instruction uses **CHARACTER** as a temporary storage area to store each character of ⟨*string*⟩. Thus, any data in **CHARACTER** prior to a **STORE** ⟨*string*⟩ instruction will be destroyed.

It is possible to encounter the end-of-the-text buffer when using this instruction. Therefore, it is recommended that this instruction include the optional syntax. Refer to the **STORE CHARACTER** construct for the semantics of this syntax.

STORE SEQUENCE

Providing the station is in sequence mode (i.e., **SEQUENCE** is **TRUE**), the **STORE SEQUENCE** construct causes the current value of the sequence number to be stored in message word [5].[26:27] as a binary integer, and message word [5].[27.1] is set **TRUE** to indicate its presence. If the station is not in sequence mode (i.e., **SEQUENCE** is **FALSE**), then the instruction is a no-op. If no message space is present at the time of the **STORE**, then an implicit ⟨*getspace instruction*⟩ is executed first.

STORE TALLY [⟨*tally number*⟩ **]**

This form causes the TALLY specified to be stored in the message space header. If no message space is present, an implicit ⟨*getspace statement*⟩ is executed just prior to the store operation.

STORE TOG [⟨*toggle number*⟩ **]**

This form causes the **TOGGLE** specified to be stored in the message space header. If no message space is present, an implicit ⟨*getspace statement*⟩ is executed just prior to the store operation.

STORE TOGS

This form stores **TOGS [0 - 7]** into the current message.

Pragmatics

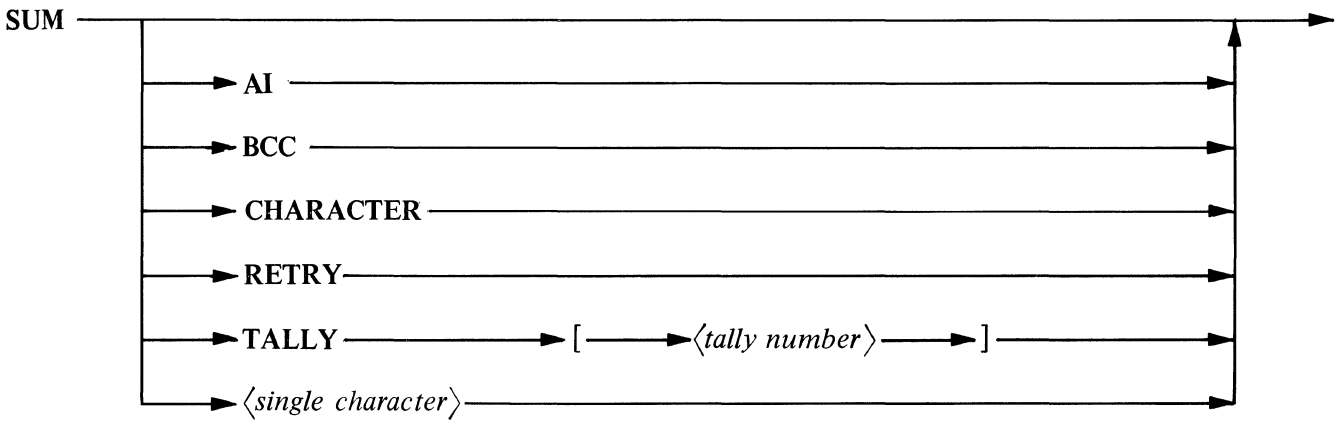
The application of the **STORE TALLY** and **STORE TOG** constructs rests solely with the programmer. Since the message space is usually returned to a Message Control System (MCS), some mutual convention could be established between the NDL programmer and the MCS programmer as to the meaning of the contents of the **TALLY**s and **TOGGLE**s.

NOTE

Only **TALLY [0] - TALLY [2]**, and **TOG [0] - TOG [7]**, can be **INITIALIZE**d from an MCS message **HEADER**, or **STORE**d there.

SUM STATEMENT

Syntax



Examples

```
SUM AI.
SUM CHARACTER.
SUM "A".
SUM TALLY [1].
```

Semantics

The purpose of the ⟨*sum statement*⟩ is to affect the calculation of the horizontal parity check (whether that be a Block Check Character or a Cyclic Redundancy Check). The specific effect of the ⟨*sum statement*⟩ is dependent upon two factors: The **SUM**med item, and whether the station's ⟨*terminal definition*⟩, for which ⟨*request definition*⟩ is running, defines horizontal parity as **CRC(16)**. Following is a description of the effect that each form of the ⟨*sum statement*⟩ has on the calculation of the horizontal parity check.

SUM

Semantically equivalent to **SUM AI**.

SUM AI.

If the horizontal parity check is a Block Check Character or is undefined, the contents of **AI** are exclusively OR-ed with the contents of **BCC**, and the result is stored in **BCC**.

If the horizontal parity check is a Cyclic Redundancy Check, the Cyclic Redundancy Check algorithm is computed with the contents of **AI** and **CRC**, and the result is stored in **CRC**.

SUM BCC

If the horizontal parity check is a Block Check Character or is undefined, then the contents of **BCC** are exclusively OR-ed with itself, and the result is stored in **BCC**. (The result in BCC would be zero in this case.)

If the horizontal parity check is a Cyclic Redundancy Check, the Cyclic Redundancy Check algorithm is computed with the contents of **CRC[0]** and **CRC**, and the result is stored in **CRC**.

SUM CHARACTER

If the horizontal parity check is a Block Check Character or is undefined, the contents of **CHARACTER** are exclusively OR-ed with the contents of **BCC**, and the result is stored in **BCC**.

If the horizontal parity check is a Cyclic Redundancy Check, the Cyclic Redundancy Check algorithm is computed with the contents of **CHARACTER** and **CRC**, and the result is stored in **CRC**.

SUM RETRY

If the horizontal parity check is a Block Check Character or is undefined, the contents of **RETRY** are exclusively OR-ed with the contents of **BCC**, and the result stored in **BCC**.

If the horizontal parity check is a Cyclic Redundancy Check, the Cyclic Redundancy Check algorithm is computed with the contents of **RETRY** and **CRC**, and the result is stored in **CRC**.

SUM TALLY [⟨*tally number*⟩]

If the horizontal parity check is a Block Check Character or is undefined, the contents of **TALLY** [⟨*tally number*⟩] are exclusively OR-ed with the contents of **BCC**, and the result is stored in **BCC**.

If the horizontal parity check is a Cyclic Redundancy Check, the Cyclic Redundancy Check algorithm is computed with the contents of **TALLY** [⟨*tally number*⟩] and **CRC**, and the result is stored in **CRC**.

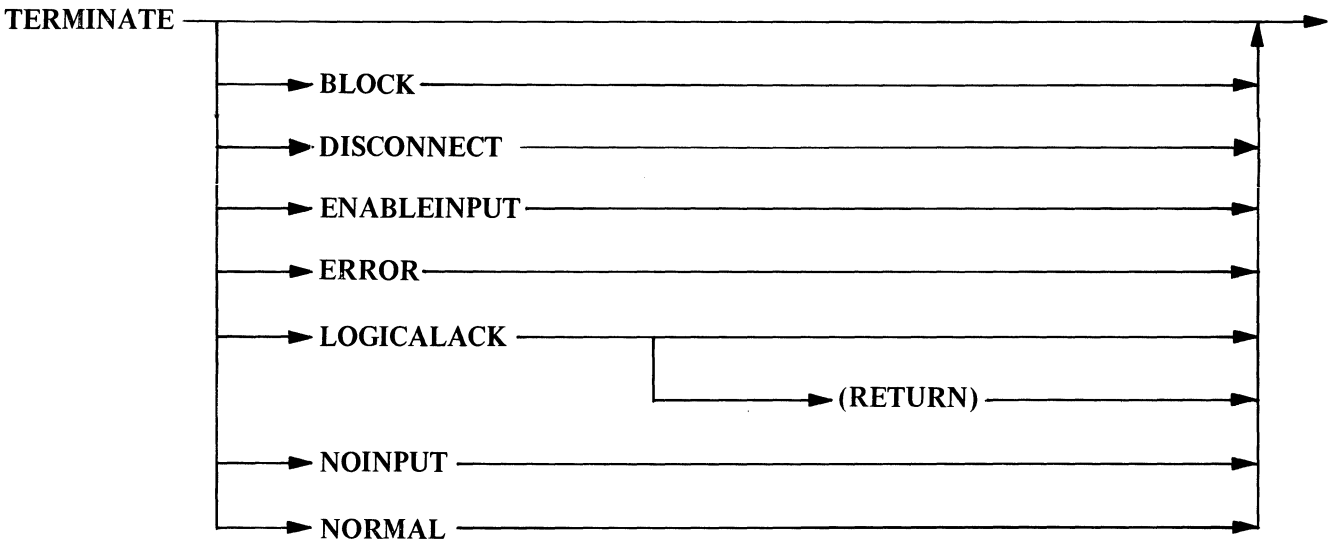
SUM ⟨*single character*⟩

If the horizontal parity check is a Block Check Character or is undefined, the ⟨*single character*⟩ is exclusively OR-ed with the contents of **BCC**, and the result is stored in **BCC**.

If the horizontal parity check is a Cyclic Redundancy Check, the Cyclic Redundancy Check algorithm is computed with the ⟨*single character*⟩ and **CRC**, and the result is stored in **CRC**.

TERMINATE STATEMENT

Syntax



Examples

```
TERMINATE NORMAL.
TERMINATE LOGICALACK.
TERMINATE LOGICALACK(RETURN).
TERMINATE.
```

Semantics

Each form of the ⟨*terminate statement*⟩ is described in the following paragraphs.

TERMINATE

This construct causes control to branch from a ⟨*request definition*⟩ and to begin executing the appropriate ⟨*control definition*⟩. Any message that may be queued is left in the Station Queue (regardless of whether the message is incoming or outgoing) and **STATION(QUEUED)** is untouched.

TERMINATE BLOCK

In a Receive Request, this construct causes the following actions:

a. an implicit ⟨*getspace instruction*⟩ is executed (in case the ⟨*request definition*⟩ may have been terminated without ever having acquired a message space);

b. the Error Flag field, Last Flag Set field, and DCP RETRY are stored into the appropriate message fields;

c. the "More-Blocks-to-Follow" bit (programmatically referenced by BLOCK) in the message (message word [0].[29:1]) is set **TRUE**;

d. the message is delinked from the Station Queue and linked into the DCP Result Queue; and

e. control continues at the next sequential statement.

In a Transmit Request, the **TERMINATE BLOCK** construct causes the following:

a. the Error Flag field, Last Flag Set field, and DCP RETRY are stored into the appropriate message fields;

b. the message is linked into the DCP Result Queue;

c. the WRITEREADY bit in the Line Table is set;

d. if STATION(QUEUED) is TRUE, each message is examined until an output message is reached or STATION(QUEUED) becomes FALSE. Messages that are not output messages are returned as ERROR RESULT messages. If an output message is reached the request definition is continued at the next sequential statement; otherwise, the request definition is suspended and put in a "sleep" state until STATION(QUEUED) becomes TRUE. When this occurs, the station queue is examined as described above.

The WRITEREADY bit is inaccessible to the NDL programmer, and it exists for each logical line in its Line Table. It is set to TRUE when a station's Transmit Request has executed the TERMINATE BLOCK construct. The WRITEREADY bit is set to FALSE when the request receives an output message to perform. Any other type of message in the station queue will return an ERROR RESULT (CLASS=99) message indicating a STATUS ERROR in the Result Byte Index to the MCS (see TERMINATE ERROR for the format of the message).

TERMINATE DISCONNECT

This construct is valid for Requests only and causes no action if the line is not a **DIALIN** (switched) type.

Execution of this construct for a **DIALIN** line causes the following actions:

a. In a Transmit Request, an error message will be returned with a Result Byte Index indicating a Transmit Abort (see **TERMINATE ERROR** for the format of the message).

b. In a Receive Request, an error message indicating a Receive Abort in the Result Byte Index is returned (see **TERMINATE ERROR** for the format of the message).

c. The line is left not ready

d. The line is disconnected, if possible, in which case a **SWITCHED STATUS RESULT (CLASS = 7)** message is returned to the **MCS**. Refer to section 7 of the B 5000/B 6000/B 7000 Series **DCALGOL** Reference Manual for information regarding this message.

TERMINATE ENABLEINPUT

This construct is allowed in Transmit Requests only.

This instruction causes the following actions:

a. the **STATION(ENABLED)** bit is tested; if **STATION(ENABLED)** is **FALSE**, then this instruction acts as a no-op; otherwise, steps b and c are executed;

b. the Error Flag field, Last Flag Set field, and DCP RETRY are stored into the appropriate message fields; and

c. control leaves the Transmit Request and the station's Receive Request is entered.

TERMINATE ERROR

This construct serves to inform the station's MCS of an unsuccessful attempt to complete a Receive or Transmit Request. This instruction inhibits the initiation of any new functions for the station.

Terminate Statement – Continued

The result of the **TERMINATE ERROR** construct is as follows:

a. **STATION(READY)** ⟨*bit variable*⟩ is set **FALSE**;

b. a minimum-size message space is obtained, filled with error information for the MCS, and linked into the DCP Result Queue (its destination being the MCS); and

c. the line is idle until the MCS takes some action.

The error message sent to the MCS contains the following information:

MSG[0].[47:8] = 99.

[39:8] = **AC** Register contents.

[31:8] = **AI** Register contents.

[23:24] = Logical Station Number.

MSG[1].[47:8] = Result Byte Index

[39:6] = Line status prior to **TERMINATE ERROR.**

[33:1] = **LINE(TOG[1]).**

[32:1] = **LINE(TOG[0]).**

[31:8] = Last Flag Set in MSG[1].[23:24]

[23:24] = Error Flag field.

MSG[2].[47:8] = **CHARACTER.**

[39:16] = Last DCP "Sleep" address.

MSG[4].[23:24] = Original DCWRITE TYPE. (Contains the original contents of MSG[0].[47:24] prior to presentation of the message to the DCP.)

Refer to the B 5000/B 6000/B 7000 Series DCALGOL Reference Manual for more information regarding this message.

TERMINATE LOGICALACK

This construct is allowed in Receive Requests only. This instruction tests the **LOGICALACK** bit in the Station Table. (The semantics of the ⟨*station logicalack statement*⟩ describe how the **LOGICALACK** bit is set.) If **LOGICALACK** is **FALSE**, the instruction acts as a no-op and control continues at the next sequential statement. If **LOGICALACK** is TRUE, the following occurs:

a. an implicit ⟨*getspace statement*⟩ is executed (in case the ⟨*request definition*⟩ is terminating without ever having acquired any message space);

b. the ACKNOWLEDGEREADY bit in the Line Table is set (the consequences of this action are described subsequently);

c. the "Message to be ACKNOWLEDGEd" bit is set in the Error Flag Field;

d. the Error Flag Field, Last Flag Set Field, and DCP RETRY are stored into the appropriate message fields;

e. the message is delinked from the Station Queue and linked into the DCP Result Queue;

f. the line is put in a "sleep" state until the station's MCS responds to the message with an ACKNOWLEDGE (TYPE = 44) DCWRITE; and

g. upon receipt of the ACKNOWLEDGE, the Receive Request is allowed to continue at the next sequential statement.

The ACKNOWLEDGEREADY bit is inaccessible to the NDL programmer, and it exists for each logical line in its Line Table. The only time that this bit will be **TRUE** is when a station's **LOGICALACK** bit is **TRUE** and its Receive Request has executed the **TERMINATE LOGICALACK** construct or the **TERMINATE LOGICALACK(RETURN)** construct. Once **TRUE**, the ACKNOWLEDGEREADY bit will not be set **FALSE**, and the ⟨*request definition*⟩ will not be allowed to continue until the MCS executes the ACKNOWLEDGE (TYPE = 44) DCWRITE.

TERMINATE LOGICALACK(RETURN)

This construct is allowed in Receive Requests only. This instruction tests the **LOGICALACK** bit in the Station Table. (The semantics of the ⟨*station logicalack statement*⟩ describe how this bit gets set.) If this bit is found **TRUE**, this statement functions exactly as does **TERMINATE LOGICALACK**; refer to that form for semantics. If **LOGICALACK** is **FALSE**, the following occurs:

a. an implicit ⟨*getspace statement*⟩ is executed (in case the ⟨*request definition*⟩ is terminating without ever having acquired a message space);

b. the Error Flag field, Last Flag Set field, and DCP RETRY are stored into the appropriate message fields;

c. the message is delinked from the Station Queue and linked into the DCP Result Queue; and

d. control continues at the next sequential instruction in the ⟨*request definition*⟩.

TERMINATE NOINPUT

If executed in a Transmit Request, this form is semantically equivalent to the **TERMINATE** construct (refer to that construct for semantics). When executed in a Receive Request, the following occurs:

a. any message space that may have been acquired is discarded;

b. **LINE(BUSY)** is set **FALSE**; and

c. control branches to the appropriate ⟨*control definition*⟩.

TERMINATE NORMAL

The purpose of this construct is to signal the satisfactory completion of a ⟨*request definition*⟩. If executed in a Receive Request, the following occurs:

a. an implicit ⟨*getspace statement*⟩ is executed (in case the ⟨*request definition*⟩ is terminating without having ever acquired a message space);

b. the Error Flag field, Last Flag Set field, and DCP ENTRY are stored into the appropriate message fields;

c. the message space is delinked from the Station Queue and linked to the DCP Result Queue;

d. **LINE(BUSY)** is set **FALSE**; and

e. control branches from the ⟨*request definition*⟩ and (providing the DCP does not take advantage of **LINE(BUSY)** set **FALSE** to initiate a ⟨*request definition*⟩) the appropriate ⟨*control definition*⟩ is entered.

Terminate Statement – Continued

If **TERMINATE NORMAL** is executed in a Transmit Request, the following occurs:

a. the Error Flag field, Last Flag Set field, and DCP RETRY are stored into the appropriate message fields;

b. the message is linked into the DCP Result Queue;

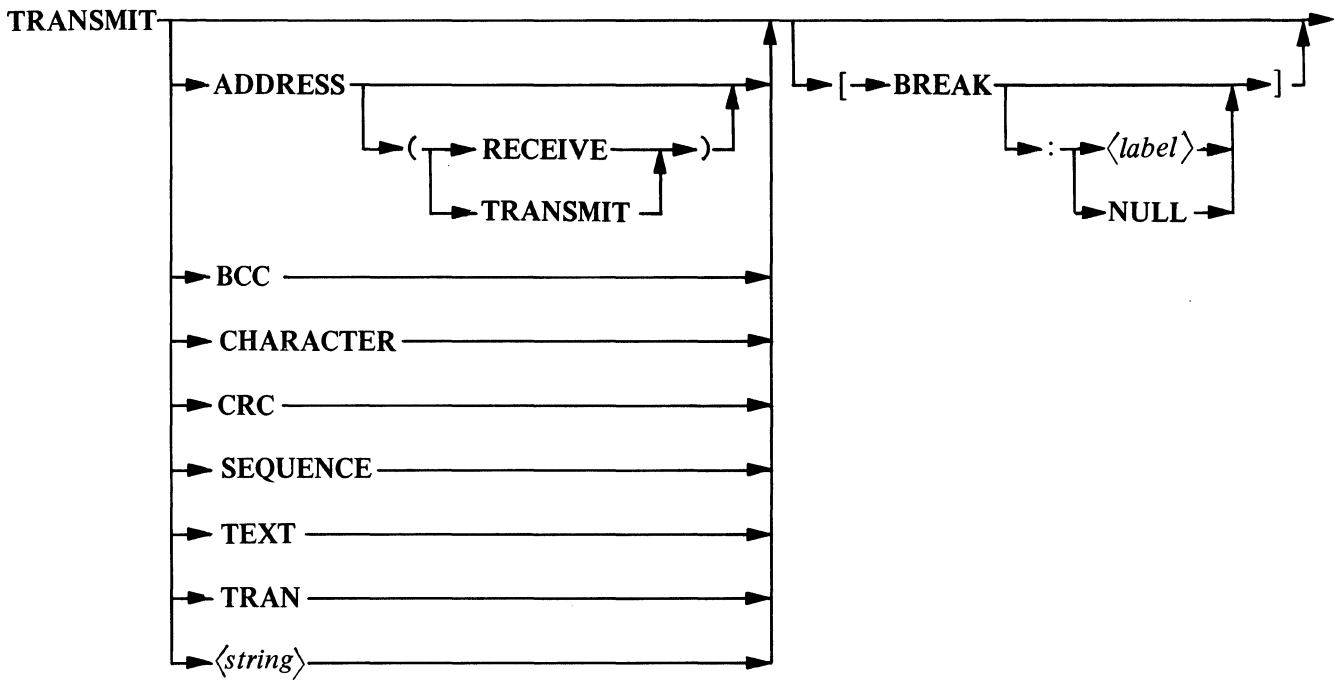
c. **LINE(BUSY)** is set **FALSE**; and

d. control branches from the ⟨*request definition*⟩ and (providing the DCP does not take advantage of **LINE BUSY**) set **FALSE** to initiate a ⟨*request definition*⟩ the appropriate ⟨*control definition*⟩ is entered.

In the Transmit Request case, the message linked to the DCP Result Queue is a result message (specifically, a GOOD RESULTS (CLASS = 5) Message). The intended destination is the MCS; however, the MCS has the option of whether to accept GOOD RESULTS Messages or to have the DCC discard them.

TRANSMIT STATEMENT

Syntax



Examples

```
TRANSMIT.
TRANSMIT CHARACTER [BREAK:NULL].
TRANSMIT SOH STX 4"00" [BREAK:10].
TRANSMIT TRAN.
TRANSMIT ADDRESS(TRANSMIT)[BREAK].
TRANSMIT TEXT[BREAK].
TRANSMIT "LITERAL STRING".
```

Semantics

The ⟨*transmit statement*⟩ causes the adapter cluster to transmit information to a terminal. The following group of syntax options specifies the information to be transmitted. All options, except **CHARACTER**, use the ⟨*byte variable*⟩ **CHARACTER** as a temporary storage area; thus, any information contained in **CHARACTER** before execution of the ⟨*transmit statement*⟩ shall be destroyed by the ⟨*transmit statement*⟩. If none of the first group of options are chosen, it is semantically equivalent to specifying **CHARACTER** (i.e., **TRANSMIT** is equivalent to **TRANSMIT CHARACTER**).

ADDRESS

The proper number of characters (as specified by the station's ⟨*terminal definition*⟩ in the ⟨*terminal address size statement*⟩) are taken from the Address field in the Station Table and transmitted.

ADDRESS(RECEIVE)

This option is equivalent to **ADDRESS**, except that **ADDRESS(RECEIVE)** must be used when an address pair is defined in the ⟨*station address statement*⟩ and the programmer wants to transmit the receive address.

ADDRESS(TRANSMIT)

This option is equivalent to **ADDRESS**, except that **ADDRESS(TRANSMIT)** must be used when an address pair is defined in the ⟨*station address statement*⟩ and the programmer wants the transmit address transmitted.

BCC

The **BCC** option causes the content of the ⟨*byte variable*⟩ **BCC** to be transmitted.

CHARACTER

The **CHARACTER** option causes the content of the ⟨*byte variable*⟩ **CHARACTER** to be transmitted.

CRC

This option causes two bytes to be transmitted; **CRC[0]** is transmitted first, followed by **CRC[1]**. If the station's ⟨*terminal definition*⟩ does not define horizontal parity as **CRC(16)**, the use of this option causes a syntax error to be generated at compile time.

SEQUENCE

This option causes the character representation of the value stored in the Sequence field of the Station Table to be transmitted if the station is in sequence mode (i.e., the ⟨*bit variable*⟩ **SEQUENCE** is **TRUE**); otherwise, the ⟨*transmit statement*⟩ is a no-op.

TEXT

This option extracts characters, one at a time, from the associated message, using **CHARACTER** as a temporary storage area, and transmits the characters until the end of the text buffer is encountered. At that point, control branches to the next statement. The **TRANSMIT TEXT** construct is, in effect, the same as the following loop:

```
1:  FETCH [ENDOFBUFFER:2].
    TRANSMIT CHARACTER.
    GO TO 1.
2:
    .
    .
    .
```

This option can only be used with the ⟨*transmit statement*⟩ in Transmit Requests.

TRAN

The proper number of transmission number characters (as defined by the station's ⟨*terminal definition*⟩ in the ⟨*terminal transmission number length statement*⟩) are extracted from the Transmit Transmission Number field in the Station Table and then transmitted.

⟨*string*⟩

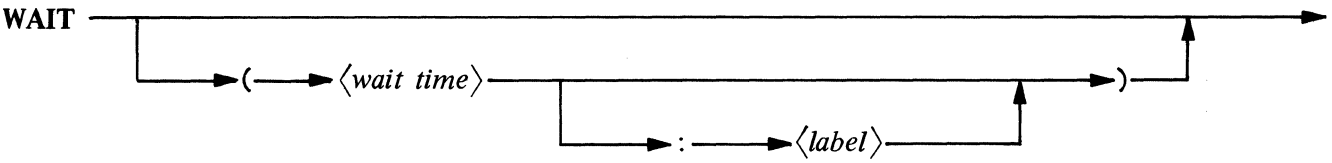
Each character of ⟨*string*⟩, using **CHARACTER** as a temporary storage area, is transmitted.

The **BREAK** option allows the programmer to specify action if a "break" is received from the terminal while the adapter cluster is still transmitting. If this option is omitted and a break occurs, an implicit **TERMINATE ERROR** instruction is executed. The following describes the actions of the three syntactical forms:

BREAK	sets **TRUE** the ⟨*bit variable*⟩ **BREAK[TRANSMIT]** and causes a branch of control to the next statement.
BREAK: ⟨*label*⟩	sets **TRUE** the ⟨*bit variable*⟩ **BREAK[TRANSMIT]** and causes a branch of control to ⟨*label*⟩.
BREAK:NULL	sets **TRUE** the ⟨*bit variable*⟩ **BREAK[TRANSMIT]**. Execution proceeds as if the break did not occur.

WAIT STATEMENT

Syntax



Examples

```
WAIT.
WAIT (3 SEC).
WAIT (5 MILLI:6).
```

Semantics

The ⟨*wait statement*⟩ is only allowed in ⟨*request definition*⟩*s* that are written to communicate with full duplex terminal types. Execution of this statement causes the ⟨*request definition*⟩ to be suspended until the co-line executes a ⟨*continue statement*⟩. The optional syntax effects the statement as described below.

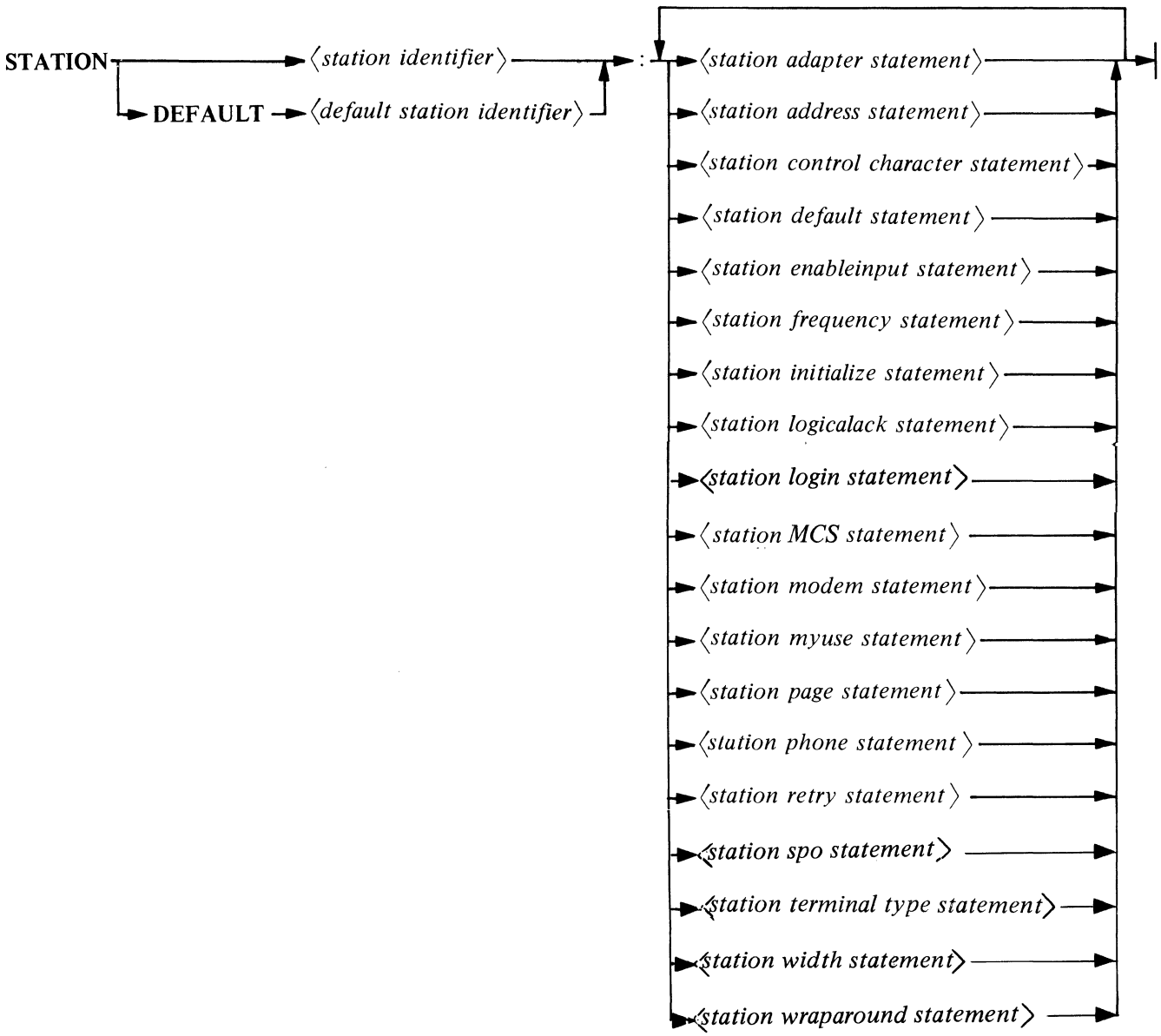
⟨*wait time*⟩	defines the maximum amount of ⟨*time*⟩ that the ⟨*request definition*⟩ should be suspending waiting for the ⟨*continue statement*⟩. If ⟨*wait time*⟩ is exceeded and the co-line has not executed a ⟨*continue statement*⟩, execution resumes at the next sequential statement.
⟨*wait time*⟩:⟨*label*⟩	same as above except execution resumes at ⟨*label*⟩ if a ⟨*continue statement*⟩ is not executed within ⟨*wait time*⟩.

Pragmatics

Refer to the ⟨*fork statement*⟩ pragmatics.

STATION DEFINITION

Syntax



Examples

```
STATION KMET:

    ENABLEINPUT  = FALSE.
    MCS          = SYSTEM/CANDE.
    CONTROL      = 4"6F".
    RETRY        = 15.
    LOGICALACK   = FALSE.
    MYUSE        = INPUT, OUTPUT.
    TERMINAL     = TELETYPE.
```

STATION DEFAULT STADFLT2:

```
CONTROL    = "?".
MCS        = SYSTEM/CANDE.
ADAPTER    = 4.
DEFAULT    = STADFLT1.
```

Semantics

⟨*station identifier*⟩ and ⟨*default station identifier*⟩ have the syntactical form of a ⟨*system identifier*⟩. Each syntactical form of the ⟨*station definition*⟩ is described subsequently.

STATION ⟨*station identifier*⟩: . . .

This form of the ⟨*station definition*⟩ defines the attributes of a station. The attributes must be defined in one of the following ways:

a. Each attribute is explicitly defined by means of a ⟨*station statement*⟩.

b. Each attribute is defined implicitly by means of an explicit reference to a set of previously defined default attribute values.

c. Some of the attributes are defined implicitly as in b, and the remainder are defined explicitly as in a.

Some of the station attributes must be defined for each station; others do not. Some of the statements may or may not be required, depending upon the appearance of other statements. The semantics portion of each ⟨*station statement*⟩ states, among other things, whether the attribute must be defined and its effect upon the requirements of other ⟨*station statement*⟩*s*.

To define the attributes of a station as described in item a above, only this syntax form is used.

To define the attributes of a station as described in items b and c above, this syntax form, the following syntax form, and the ⟨*station default statement*⟩ must be used in conjunction (this is described under the following syntax form).

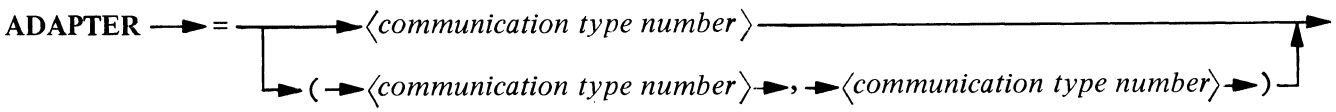
STATION DEFAULT ⟨*default station identifier*⟩: . . .

This form is referred to as a Default ⟨*station identifier*⟩. Its purpose is to decrease the number of source statements required to define all of the stations. This is accomplished in the following manner. Attributes common to several stations are defined by means of a Default ⟨*station definition*⟩. Associated with each Default ⟨*station definition*⟩ is a ⟨*default station identifier*⟩. Subsequent to the Default ⟨*station definition*⟩, any ⟨*station definition*⟩ can reference the ⟨*default station identifier*⟩, instead of repeating the list. A ⟨*default station identifier*⟩ is referenced by means of a ⟨*station default statement*⟩. The NDL compiler uses the last definition of a station attribute, and therefore the programmer can reference a Default ⟨*station definition*⟩ and change any attributes by redefining them in the ⟨*station definition*⟩.

In appearance, the Default ⟨*station definition*⟩ is similar to the ⟨*station definition*⟩. The differences are that the reserved word DEFAULT follows the reserved word STATION, and that there are no statements that are required to appear in a Default ⟨*station definition*⟩.

STATION ADAPTER STATEMENT

Syntax



Examples

ADAPTER = 4.
ADAPTER = (11,6).

Semantics

The ⟨*station adapter statement*⟩ defines a combination of character format, synchronous/asynchronous communication, and line speed (in the case of synchronous communications) that the DCP must use to communicate with the terminal associated with the station. This is done by supplying a ⟨*communication type number*⟩ (or number pair). Table 5-4 lists the allowed ⟨*communication type number*⟩*s* and the characteristics associated with each.

For example,

ADAPTER = 4.

This statement defines an 11-bit character format, asynchronous communication, at a line speed of 110 bits per second.

If the station's associated terminal type utilizes full duplex (i.e., the ⟨*terminal duplex statement*⟩ specifies **DUPLEX=TRUE**), and the primary and the auxiliary lines have different characteristics, then a ⟨*communication type number*⟩ pair must be supplied.

For example,

ADAPTER = (11,6).

This statement defines for the primary line a 10-bit character format, asynchronous communication, at a speed of 1800 bits per second. The characteristics associated with the auxiliary line are the same except that it runs at a line speed of 150 bits per second.

The statement:

ADAPTER = (6,6).

is semantically equivalent to:

ADAPTER = 6.

The ⟨*communication type number*⟩ (or number pair) defined in this statement must be one of those listed in the ⟨*terminal adapter statement*⟩ of the station's associated ⟨*terminal definition*⟩. The ⟨*station adapter statement*⟩ is required unless the ⟨*terminal adapter statement*⟩ lists only one ⟨*communication type number*⟩ (or number pair), in which case, the ⟨*station adapter statement*⟩ may be omitted and the ⟨*terminal adapter statement*⟩ specification is used.

Supplementary Examples

The following program fragments illustrate valid adapter statement specifications.

Example 1

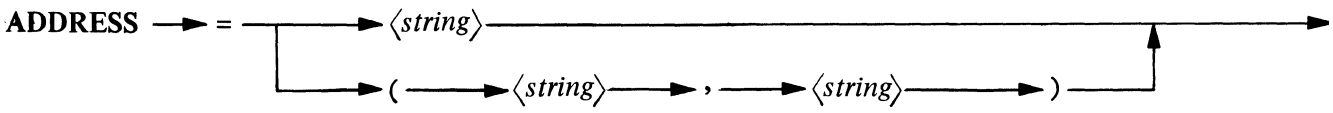
```
MODEM AMODEM:
    ADAPTER=1,2,3,4,5,6,7,8,9,10.
        .
        .
        .
TERMINAL ATERMINAL:
    ADAPTER=6,7,8,9,11,12,13,14,15.
        .
        .
        .
STATION ASTATION:
    ADAPTER=7.
    MODEM=AMODEM.
    TERMINAL=ATERMINAL.
        .
        .
        .
```

Example 2

```
MODEM DUPLEXMODEM:
    ADAPTER=6, (11,6), (12,6), 12.
        .
        .
        .
TERMINAL DUPLEXTERMINAL:
    ADAPTER=6, (11,6).
        .
        .
        .
STATION DUPLEXSTATION:
    ADAPTER=(11,6).
    MODEM=DUPLEXMODEM.
    TERMINAL=DUPLEXTERMINAL.
        .
        .
        .
```

STATION ADDRESS STATEMENT

Syntax



Examples

```
ADDRESS = 4"01".
ADDRESS = (4"0001",4"01").
ADDRESS = "A".
```

Semantics

The ⟨*station address statement*⟩ defines the actual address characters of the station's terminal that are required for operations such as polling and selecting. The number of characters in the ⟨*string*⟩(*s*) must be equal to the number defined in the ⟨*terminal address size statement*⟩ of the associated terminal. This statement is not allowed in Default ⟨*station definitions*⟩.

ADDRESS = ⟨*string*⟩.

This form of the statement is used when the receive address and the transmit address are the same.

ADDRESS = (⟨*string*⟩, ⟨*string*⟩).

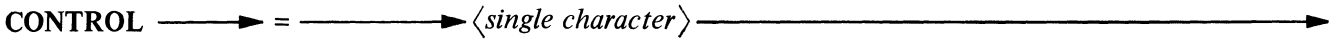
This form of the statement must be used if the receive address and the transmit address differ. The first ⟨*string*⟩ defines the receive address characters, and the second ⟨*string*⟩ defines the transmit address characters.

Pragmatics

The address characters of a station can be changed as a result of the Message Control System (MCS) executing a SET CHARACTERS (TYPE = 39) DCWRITE.

STATION CONTROL CHARACTER STATEMENT

Syntax



Example

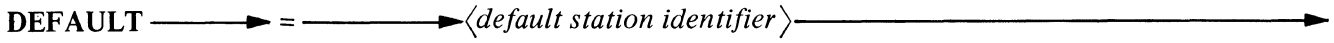
```
CONTROL = "?".
```

Semantics

The ⟨*station control character statement*⟩ defines the control character of the station. The control character can be recognized by the DCP when **RECEIVE**d in a message text from the station, and any action to be taken can be specified by the programmer using the **CONTROL** syntax in the ⟨*receive statement*⟩.

STATION DEFAULT STATEMENT

Syntax



Example

```
DEFAULT = STADFLT1.
```

Semantics

The ⟨*station default statement*⟩ allows the programmer to specify the ⟨*default station identifier*⟩ of a set of previously defined default station attributes to be used for a ⟨*station definition*⟩ whose description is incomplete. It is advantageous to group common attributes under a Default ⟨*station definition*⟩ and list the remaining attributes under each individual ⟨*station definition*⟩. The compiler will then refer to the Default ⟨*station definition*⟩ to complete the ⟨*station definition*⟩. The ⟨*station default statement*⟩ is not required to appear in each ⟨*station definition*⟩; however, a ⟨*station definition*⟩ must define all required attributes locally if a ⟨ *tation default statement*⟩ does not appear.

The ⟨*station default statement*⟩ can appear in a ⟨*station definition*⟩ or a Default ⟨*station definition*⟩.

Supplementary Example

The following is an example of how a Default ⟨*station definition*⟩ can be used in conjunction with a ⟨*station definition*⟩.

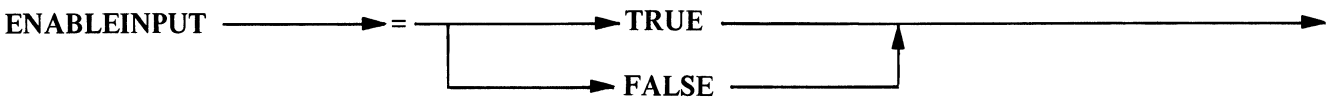
```
STATION DEFAULT STADFLT:
    MCS           = SYSTEM/CANDE.
    CONTROL       = "?".
    RETRY         = 15.
    LOGICALACK    = FALSE.
    MYUSE         = INPUT, OUTPUT.
    TERMINAL      = TELETYPE.
    ENABLEINPUT   = TRUE.

STATION TESTSTATION:
    DEFAULT       = STADFLT.
    MODEM         = MABELL103A.
    MCS           = SYSTEM/DIAGNOTICMCS.
    ADAPTER       = 4.
```

{Default ⟨*station definition*⟩}

{⟨*station default statement*⟩ references Default ⟨*station definition*⟩ above to complete the ⟨*station definition*⟩.}

STATION ENABLEINPUT STATEMENT

Syntax



Semantics

The ⟨*station enableinput statement*⟩ defines the initial state of the station's "enabled" bit (programmatically referred to as **STATION(ENABLED)**). This statement must be defined in each ⟨*station definition*⟩.

ENABLEINPUT = TRUE.

This construct causes the "enabled" bit to be initially **TRUE** after DCP initialization, and the station is said to be "enabled for input," or simply "enabled."

ENABLEINPUT = FALSE.

This construct causes the "enabled" bit to be initially **FALSE** after DCP initialization, and the station is said to be "disabled for input," or "disabled."

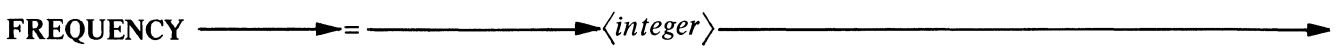
Pragmatics

Whether a station is enabled or disabled for input can directly affect the execution sequence of instructions in the ⟨*control definition*⟩ and ⟨*request definition*⟩ (*s*) designated for that station. Specifically, if the station is disabled for input, control will never branch to the Receive Request for that station as a result of either an **INITIATE ENABLEINPUT** or a **TERMINATE ENABLEINPUT** construct. Refer to the **INITIATE ENABLEINPUT** and **TERMINATE ENABLEINPUT** constructs for more detailed information.

The MCS of the station may change the state of the "enabled" bit, after DCP initialization, by means of the ENABLE INPUT (TYPE = 35) DCWRITE or the DISABLE INPUT (TYPE = 36) DCWRITE.

STATION FREQUENCY STATEMENT

Syntax



Example

```
FREQUENCY = 10.
```

Semantics

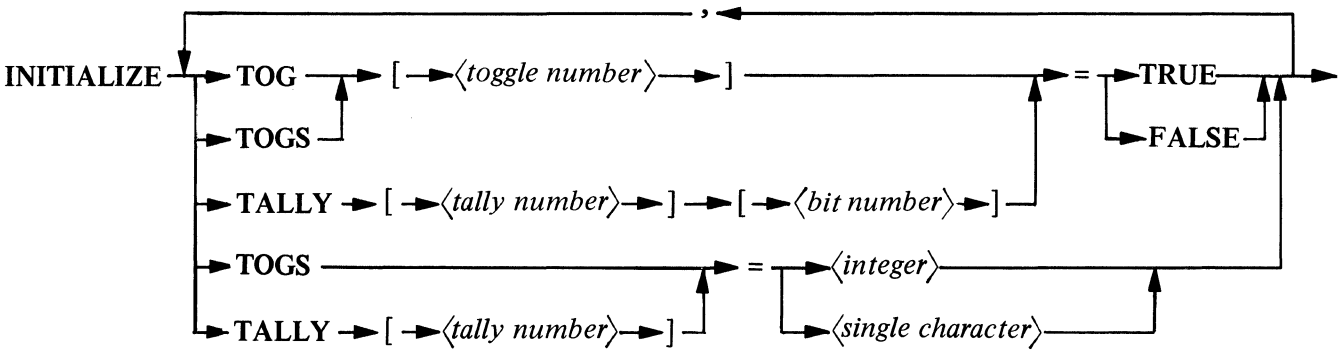
The ⟨*station frequency statement*⟩ defines the initial value of the ⟨*byte variable*⟩ programmatically referred to as **STATION (FREQUENCY)**. The ⟨*control definition*⟩ specified for the station can reference the ⟨*byte variable*⟩ and use the value stored there in any way that the programmer sees fit; however, the intended use of the variable is to influence in some way the rate at which a polled station is polled. In the polling ⟨*control definition*⟩ provided by Burroughs Corporation in SYMBOL/SOURCENDL, **FREQUENCY** specifies a relative polling rate: 0 means poll at the highest rate, 1 means to poll at a slower rate, 2 means to poll at a still lower rate, etc.

The ⟨*integer*⟩ must not exceed a value of 255.

The MCS of the defined station can change the value of **STATION(FREQUENCY)** by means of an ENABLE INPUT (TYPE = 35) DCWRITE.

STATION INITIALIZE STATEMENT

Syntax



Examples

```
INITIALIZE TOGS = 4"FF".
INITIALIZE TOG[0] = TRUE,
           TALLY[1] = 25,
           TALLY[0] [7] = TRUE.
INITIALIZE TALLY[0] = "?".
```

Semantics

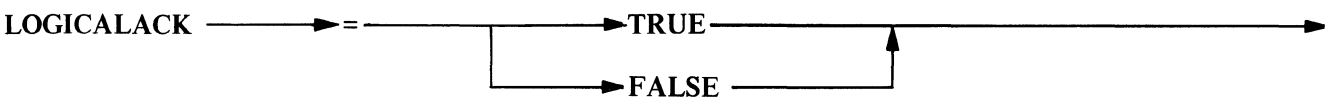
The ⟨*station initialize statement*⟩ provides the means to define initial values for the station **TOGGLE**s and **TALLY**s. Any initial values defined for station **TOGGLE**s and **TALLY**s are stored in the **TOGGLE**s and **TALLY**s at DCP initialization time only.

NOTE

Only **TALLY [0] - TALLY [2]**, and **TOG [0] - TOG [7]**, can be **INITIALIZE**d in the **STATION SECTION**.

STATION LOGICALACK STATEMENT

Syntax



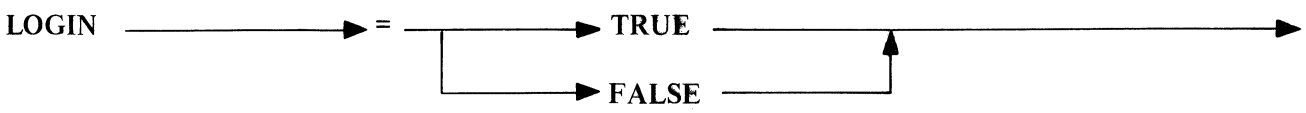
Semantics

The ⟨*station logicalack statement*⟩ defines the initial state of a bit, referred to as the Logicalack bit, in the Station Table. **TRUE** or **FALSE** can be specified, indicating the initial state as on or off, respectively. If the **LOGICALACK** bit is on, special action is taken if the Receive Request executes either the **TERMINATE LOGICALACK** or **TERMINATE LOGICALACK(RETURN)** constructs of the ⟨*terminate statement*⟩. This statement is required in ⟨*station definition*⟩*s*.

The MCS of the station can change the value of the Logicalack bit after DCP initialization by means of the SET/RESET LOGICALACK (TYPE = 43) DCWRITE.

STATION LOGIN STATEMENT

Syntax



Semantics

The ⟨*station login statement*⟩ defines the initial state of the "login" bit in the Data Comm Controller's station table. The default setting of this bit is **FALSE.**

The logical value of the bit may be interrogated by the **MCS** controlling the station. Presently, the meaning of this bit is established by convention between the **NDL** programmer and the **MCS** programmer.

STATION MCS STATEMENT

Syntax

Examples

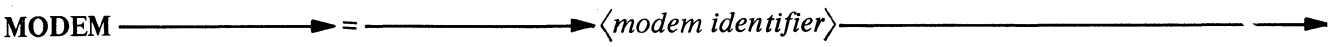
```
MCS = SYSTEM/RJE.
MCS = SYSTEM/APL.
MCS = UTILITY/MCS.
```

Semantics

The ⟨*station MCS statement*⟩ defines the Message Control System (MCS) that is responsible for handling messages to and from the station. If the MCS named is not an MCS defined in a ⟨*MCS definition*⟩, it is added to the list of valid MCS programs to be contained in the Network Information File, and the MCS will not be allowed to execute diagnostic DCWRITEs. Refer to the semantics of the ⟨*MCS definition*⟩ for information regarding the diagnostic DCWRITEs. This statement is required in ⟨*station definition*⟩*s*.

STATION MODEM STATEMENT

Syntax



Example

```
MODEM = BELL201.
```

Semantics

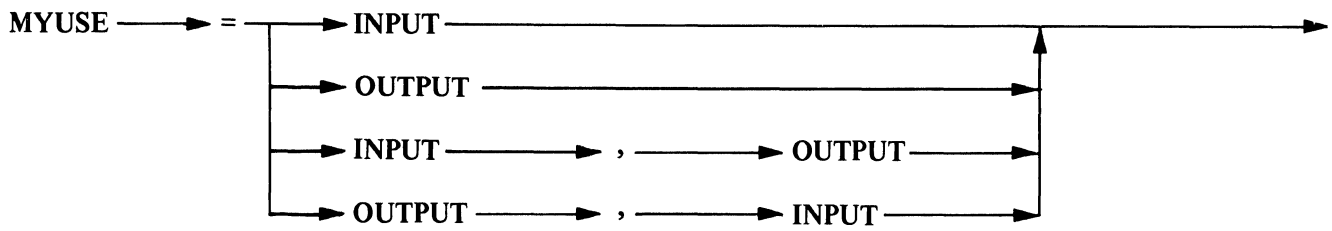
The ⟨*station modem statement*⟩ applies to a station that has associated with it a terminal type that must communicate with the Data Communications System through the use of a modem. This statement associates the modem type (i.e., a ⟨*modem definition*⟩) used for that purpose with the station. If this statement is omitted from the ⟨*station definition*⟩, and the ⟨*line definition*⟩ for the line to which the station is assigned (if, in fact, the station is assigned to a line) does not contain a ⟨*line modem statement*⟩, then the compiler assumes a direct connection between the terminal and the line adapter.

The ⟨*modem identifier*⟩ must name a ⟨*modem definition*⟩ that is compatible with the defined station attributes. To be more specific, the ⟨*communication type number*⟩ specified in the ⟨*station adapter statement*⟩ (or in the ⟨*terminal adapter statement*⟩ of the station's ⟨*terminal definition*⟩ if no ⟨*station adapter statement*⟩ appears) must be one of the ⟨*communication type number*⟩*s* listed in the ⟨*modem adapter statement*⟩ of the modem named.

After DCP initialization, the MCS of the station may change the ⟨*modem definition*⟩ associated with the station, by means of the MOVE/ADD/SUBTRACT STATION (TYPE = 130) DCWRITE.

STATION MYUSE STATEMENT

Syntax



Semantics

The ⟨*station myuse statement*⟩ defines to what extent an object job can use the station as an input or output device.

MYUSE=INPUT specifies that an object job can use the station as an input file only.

MYUSE=OUTPUT specifies that an object can use the station as an output file only.

MYUSE=INPUT,OUTPUT or **MYUSE=OUTPUT,INPUT** specifies that an object job can use the station as an input and/or output file.

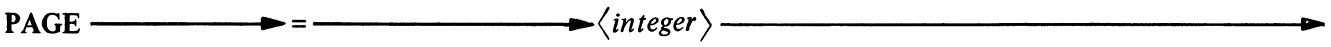
A ⟨*terminal request statement*⟩ must be defined by the station's ⟨*terminal definition*⟩ for handling input and/or output capabilities as specified in the ⟨*station myuse statement*⟩. Thus, if the station is to send input to, and receive output from, an object job, the station's ⟨*terminal definition*⟩ must specify a Transmit Request and a Receive Request.

Note that the ⟨*station myuse statement*⟩ restricts the use of the station by object jobs only. The MCS can communicate with the station to the extent specified in the ⟨*terminal request statement*⟩ of the station's ⟨*terminal definition*⟩. That is, regardless of what is specified in the ⟨*station myuse statement*⟩, the MCS can receive information from, or send information to, a station, provided that the station's ⟨*terminal definition*⟩ specified a Receive and Transmit Request.

The station **MYUSE** attribute can be interrogated by an object job through reference to the **MYUSE** file attribute. For further information, refer to the B 7000/6000 Input/Output Subsystem Information Manual, form number 5001779.

STATION PAGE STATEMENT

Syntax



Examples

```
PAGE = 12.
PAGE = 0.
```

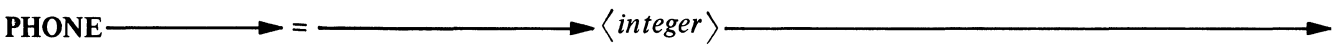
Semantics

The ⟨*station page statement*⟩ defines the number of logical lines per logical page. The ⟨*integer*⟩ specified must be less than or equal to the number of lines specified in the ⟨*terminal page statement*⟩ of the station's ⟨*terminal definition*⟩ (unless that number is zero, indicating pagination is arbitrary). If a ⟨*station page statement*⟩ is not included in the ⟨*station definition*⟩ , the station's ⟨*terminal definition*⟩ specifications for pagination are used.

An object job may obtain the **PAGE** value of a station, if the station is attached to a file, and that file is open, by interrogating the **PAGESIZE** file attribute and supplying the File Relative Station Number (FRSN). Refer to the **PAGESIZE** attribute in the B 7000/6000 Input/Output Subsystem Information Manual, form number 5001779, for more information.

STATION PHONE STATEMENT

Syntax



Example

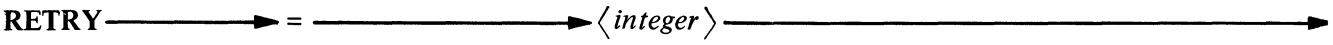
```
PHONE = 12136572385.
```

Semantics

The ⟨*station phone statement*⟩ is implemented for documentation purposes only. This statement documents the telephone number that the system would have to dial to reach the station's terminal.

STATION RETRY STATEMENT

Syntax



Example

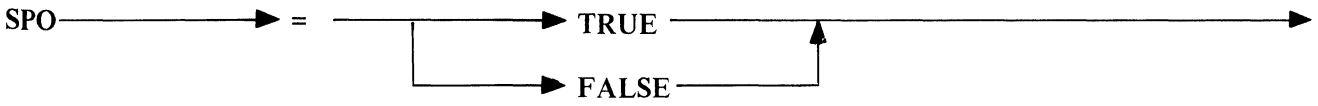
```
RETRY = 3.
```

Semantics

The ⟨*station retry statement*⟩ defines a default value for DCP INITIAL RETRY. Refer to the **RETRY** ⟨*byte variable*⟩ for more information.

STATION SPO STATEMENT

Syntax



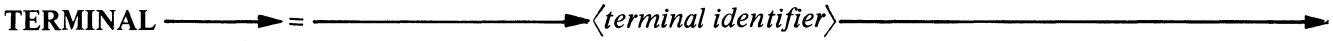
Semantics

The ⟨*station spo statement*⟩ defines the initial state of the "spo" bit in the Data Comm Controller's station table. The default setting of this bit is **FALSE**.

The value of the bit may be interrogated by the MCS controlling the station. Presently, the menaing of this bit is established by convention between the **NDL** programmer and the **MCS** programmer.

STATION TERMINAL TYPE STATEMENT

Syntax



Examples

```
TERMINAL = APLTERM.
TERMINAL = TTY.
```

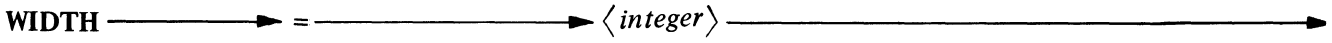
Semantics

The ⟨*station terminal type statement*⟩ associates a terminal type with the station. This statement is required in a ⟨*station definition*⟩.

After DCP initialization, the MCS of the station can change the terminal type associated with the station by means of the MOVE/ADD/SUBTRACT STATION (TYPE = 130) DCWRITE.

STATION WIDTH STATEMENT

Syntax



Examples

```
WIDTH = 72.
WIDTH = 132.
```

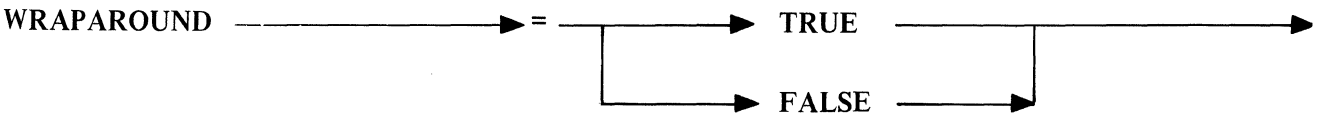
Semantics

The ⟨*station width statement*⟩ defines the number of characters in a logical display line of output on the station's terminal. If this statement is not included in a ⟨*station definition*⟩ , then the **WIDTH** defined for the station's ⟨*terminal definition*⟩ is the default station **WIDTH**.

An object job can interrogate a station's **WIDTH** by testing the value of the **WIDTH** file attribute. Refer to the B 6700 Input/Output Subsystem Information Manual, form number 5000185, for further information.

STATION WRAPAROUND STATEMENT

Syntax



Semantics

The ⟨*station wraparound statement*⟩ defines the initial state of the "wraparound" bit in the Data Comm Controller's station table. The default setting of this bit is **FALSE.**

The logical value of the bit may be interrogated by the **MCS** controlling the station. Presently, the meaning of this bit is established by convention between the **NDL** programmer and the **MCS** programmer.

TERMINAL DEFINITION

Syntax

Examples

TERMINAL TTY:

```
CODE       =  ASC67.
PARITY     =  NULL.
SCREEN     =  FALSE.
BUFFER     =  NULL.
DUPLEX     =  FALSE.
ADDRESS    =  NULL.
WIDTH      =  72.
MAXINPUT   =  72.
TIMEOUT    =  300 SEC.
REQUEST    =  RECEIVE: READTTY,
              TRANSMIT: WRITETTY.
CONTROL    =  CONTENTIONDEVICE.
```

TERMINAL DEFAULT DEFAULTLIST1:

```
CODE       =  ASC67.
PARITY     =  NULL.
SCREEN     =  FALSE.
BUFFER     =  NULL.
```

Semantics

⟨*terminal identifier*⟩ and ⟨*default terminal identifier*⟩ each have the syntactic form of ⟨*identifier*⟩.

Each construct of the ⟨*terminal definition*⟩ is described subsequently.

TERMINAL ⟨*terminal identifier*⟩ : . . .

This form of the ⟨*terminal definition*⟩ syntax defines the attributes of a terminal type in the data communications network. Most terminal attributes are hardware-dependent. The attributes of the terminal type are defined in one of the following ways:

a. Each attribute is defined explicitly by means of a ⟨*terminal attribute statement*⟩ in the ⟨*terminal definition*⟩.

b. Each attribute is defined implicitly by an explicit reference to a set of previously defined default attribute values.

c. Some of the attributes are defined implicitly as in b, and the remainder are defined explicitly as in a.

Some of the ⟨*terminal statement*⟩*s* must be defined for each ⟨*terminal definition*⟩; others do not. Some of the statements may or may not be required, depending upon the appearance of other statements. The semantics portion of each ⟨*terminal attribute statement*⟩ states, among other things, whether the attribute must be defined and its effect upon the requirement of other attribute definitions.

To define the attributes of a **TERMINAL** as described in item a above, this syntax form must be used.

To define the attributes of a terminal type as described in items b and c above, this syntax form, the following syntax form, and the ⟨*terminal default statement*⟩ must be used in conjunction (this is described under the following syntax form).

TERMINAL DEFAULT ⟨*default terminal identifier*⟩ : . . .

This form is referred to as a Default ⟨*terminal definition*⟩.

Its purpose is to decrease the number of source statements required to define all of the terminal types in the data communications system. This is accomplished in the following manner. Attributes common to several terminal types are defined by means of a Default ⟨*terminal definition*⟩. Associated with each Default ⟨*terminal definition*⟩ is a ⟨*default terminal identifier*⟩. Subsequent to the Default ⟨*terminal definition*⟩, any ⟨*terminal definition*⟩ that has those attributes in common may reference the ⟨*default terminal identifier*⟩, instead of repeating the list. (A ⟨*default terminal identifier*⟩ is referenced by means of a ⟨*terminal default statement*⟩.) The NDL compiler uses the last definition of a terminal attribute, and therefore the programmer can reference a Default ⟨*terminal definition*⟩ and change any attributes by redefining them in the ⟨*terminal definition*⟩.

In appearance, the Default ⟨*terminal definition*⟩ is similar to the ⟨*terminal definition*⟩. The differences are that the reserved word **DEFAULT** follows the reserved word **TERMINAL**, and that no statements are required to appear in a Default ⟨*terminal definition*⟩.

Supplementary Example

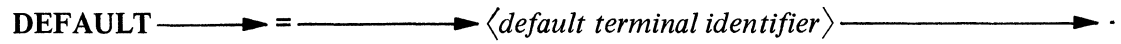
Below is an example of how a Default ⟨*terminal definition*⟩ can be used in conjunction with a ⟨*terminal definition*⟩.

```
TERMINAL DEFAULT DEFAULTLIST1:

    CODE    = ASC67.
    PARITY  = NULL.
    SCREEN  = FALSE.
    BUFFER  = NULL.
```

DEFAULTLIST1 is the ⟨*default terminal identifier*⟩ of this Default ⟨*terminal definition*⟩. The set of default attributes that follows is referenced by this name.

⟨*terminal statement*⟩*s* define the default attributes associated with DEFAULTLIST1.

Above, DEFAULTLIST1 has associated with it four attributes. Any subsequent ⟨*terminal definition*⟩ in a source program can reference these default attributes by the appearance of a ⟨*terminal default statement*⟩ in the ⟨*terminal definition*⟩. The ⟨*terminal default statement*⟩ has the form:



where the ⟨*default terminal identifier*⟩ must name a previously defined Default ⟨*terminal definition*⟩. More information regarding the use of Default ⟨*terminal definition*⟩*s* in conjunction with ⟨*terminal default statement*⟩*s* can be found in the ⟨*terminal default statement*⟩ semantics.

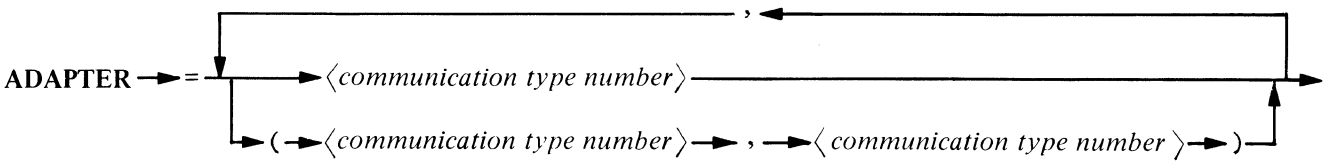
Below, TTY uses the ⟨*terminal default statement*⟩ to reference DEFAULTLIST1. DEFAULTLIST1 contains the attribute information required to complete the ⟨*terminal definition*⟩ TTY.

```
TERMINAL TTY:

    DEFAULT   = DEFAULTLIST1.
    DUPLEX    = FALSE.
    ADDRESS   = NULL.
    WIDTH     = 72.
    MAXINPUT  = 72.
    TIMEOUT   = 300 SEC.
    CONTROL   = CONTENTIONDEVICE.
    REQUEST   = RECEIVE:  READTTY, TRANSMIT: WRITETTY.
```

This ⟨*terminal default statement*⟩ references the Default ⟨*terminal definition*⟩ defined previously.

TERMINAL ADAPTER STATEMENT

Syntax



Examples

ADAPTER = 4.
ADAPTER = (6,10), (10,6).
ADAPTER = 5, (5,6), 6.

Semantics

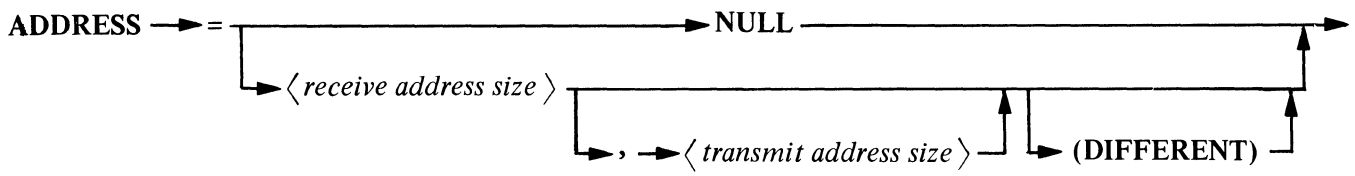
The ⟨*terminal adapter statement*⟩ defines one or more combinations of character format, synchronous/asynchronous communication, and line speed (in the case of asynchronous communications), with which the terminal type is compatible. This is done by supplying one or more ⟨*communication type number*⟩*s* (or number pairs). Table 5-4 lists the allowed ⟨*communication type number*⟩*s* and the characteristics associated with each.

If the terminal type is to be operated in a full duplex mode, and the primary and the auxiliary lines have different characteristics, then a ⟨*communication type number*⟩ pair must be supplied.

If the terminal is to be modem-connected (i.e., connected to the system through the use of modems), then at least one of the ⟨*communication type number*⟩*s* (or number pairs) must be compatible with those numbers listed for the connecting modem in the ⟨*modem adapter statement*⟩.

TERMINAL ADDRESS SIZE STATEMENT

Syntax



Examples

ADDRESS = 2.
ADDRESS = 2 (DIFFERENT).
ADDRESS = 3, 2 (DIFFERENT).
ADDRESS = 2,3.

Semantics

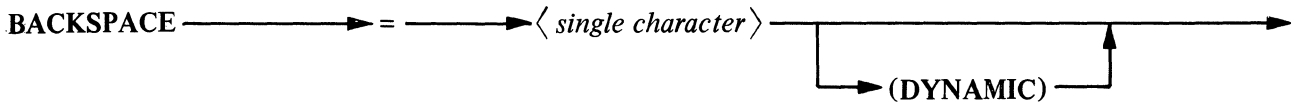
The ⟨*terminal address size statement*⟩ defines the number of address characters that the terminal type transmits and receives. The number of address characters must not be confused with the actual address characters used in polling and selecting; the ⟨*station address statement*⟩ defines the actual address characters. This attribute must be defined when actual address characters are defined in the ⟨*station address statement*⟩ of a ⟨*station definition*⟩ that references the ⟨*terminal definition*⟩.

⟨*receive address size*⟩ and ⟨*transmit address size*⟩ must be integers greater than zero and less than 4. The ⟨*receive address size*⟩ defines the number of address characters the terminal expects to receive, and the ⟨*transmit address size*⟩ defines the number of address characters that the terminal transmits. If the ⟨*transmit address size*⟩ is not defined, it is assumed the ⟨*transmit address size*⟩ is equal in length to the ⟨*receive address size*⟩. The ⟨*receive address size*⟩ and the ⟨*transmit address size*⟩ for a given terminal must concur with the length of the character ⟨*string*⟩(*s*) defined as the actual address characters in the ⟨*station address statement*⟩ of any ⟨*station definition*⟩ which references the ⟨*terminal definition*⟩; otherwise, a syntax error results.

The **(DIFFERENT)** option must be used if the ⟨*receive address*⟩ and the ⟨*transmit address*⟩, as defined in the ⟨*station address statement*⟩, are not identical.

TERMINAL BACKSPACE CHARACTER STATEMENT

Syntax



Examples

BACKSPACE = 4"16".
BACKSPACE = "←" (DYNAMIC).

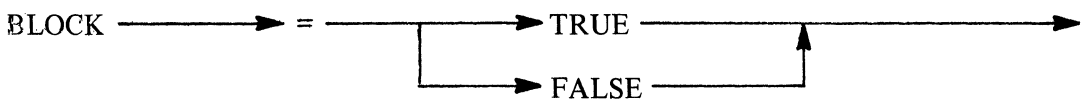
Semantics

The ⟨*terminal backspace character statement*⟩ defines the backspace character of the terminal type (i.e., the character that the terminal type would transmit to indicate that the previous character should be deleted). If defined, the backspace character can be recognized by the DCP when **RECEIVE**d (in a ⟨*receive statement*⟩), and any action to be taken can be specified by the programmer (using the **BACKSPACE** syntax).

(DYNAMIC) indicates that the controlling MCS of a station referencing the ⟨*terminal definition*⟩ is allowed to change the backspace character for the station by means of a SET CHARACTERS (TYPE=39) DCWRITE.

TERMINAL BLOCK STATEMENT

Syntax



Example

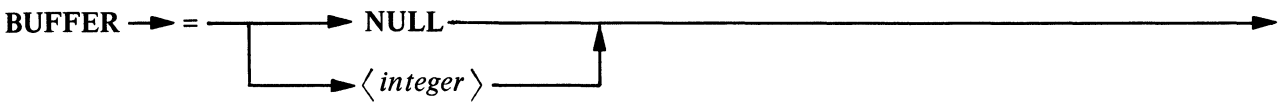
```
BLOCK = FALSE.
```

Semantics

This statement is implemented for program documentation purposes only. This statement provides a means of documenting a terminal capable of blocked input and/or output. The documentation of this function is optional in a ⟨*terminal definition*⟩.

TERMINAL BUFFER SIZE STATEMENT

Syntax



Examples

```
BUFFER = NULL.
BUFFER = 960.
```

Semantics

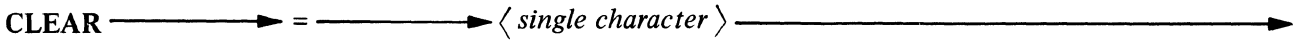
The ⟨*terminal buffer size statement*⟩ applies to buffered devices and defines the size, in characters, of the terminal type buffer. If the terminal type is an unbuffered device, the form:

```
BUFFER = NULL.
```

can be used, or the statement may be omitted. If the ⟨*terminal buffer size statement*⟩ is omitted or if **BUFFER = NULL.**, then the ⟨*terminal maxinput statement*⟩ or the ⟨*terminal maxoutput statement*⟩ must be defined for the ⟨*terminal definition*⟩.

TERMINAL CLEAR CHARACTER STATEMENT

Syntax



Example

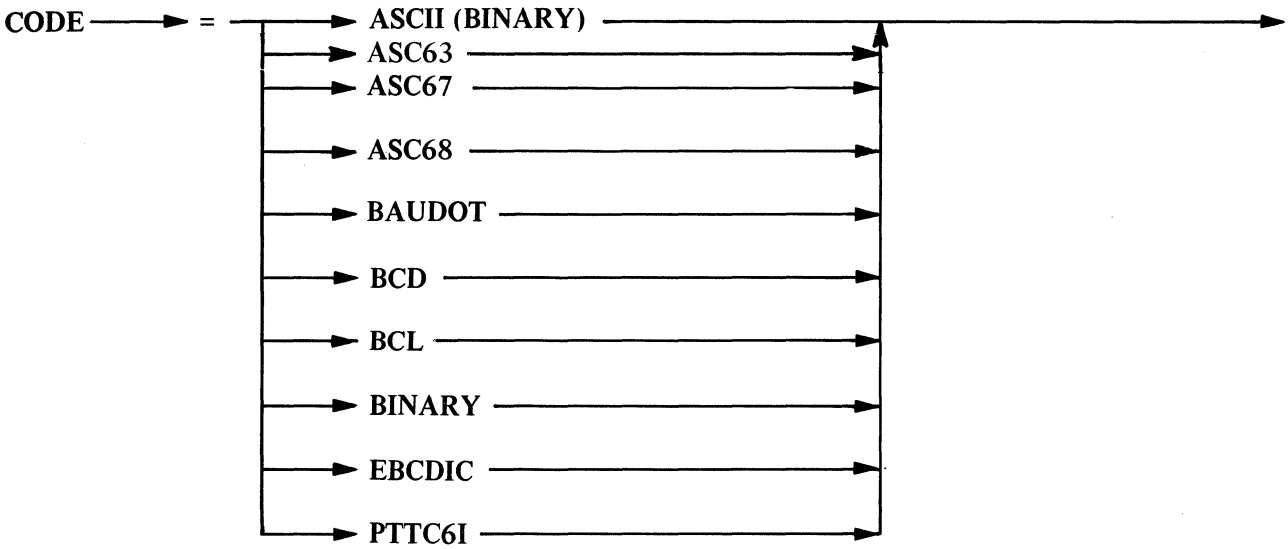
```
CLEAR = 4"11".
```

Semantics

This statement is implemented for program documentation purposes only. It provides a means to document the clear character of a terminal type. The documentation of this character is optional in a ⟨*terminal definition*⟩.

TERMINAL CODE STATEMENT

Syntax



Semantics

The ⟨*terminal code statement*⟩ specifies the character code translation required for the DCP to communicate with the terminal type. The internal code of the DCP is EBCDIC, and the DCP translates from EBCDIC to the code specified for transmissions, and from the code specified to EBCDIC for receptions.

BINARY and **EBCDIC** specify that translation is not required.

ASC63, **ASC67** and **ASC68** specify the standard software translation tables for the ASCII character code.

ASCII (BINARY) allows a ⟨*control definition*⟩ or ⟨*request definition*⟩ to switch back and forth between ASCII code translation and no translation. The ⟨*code statement*⟩ in a ⟨*request definition*⟩ or ⟨*control definition*⟩ effects the switch back and forth. The application of this feature is to allow a ⟨*request definition*⟩ or ⟨*control definition*⟩ to enter a "transparent" mode in Binary Synchronous communications procedures.

BAUDOT, BCD, BCL, and **PTTC6I** specifications are all indicative of the translation they invoke. For example, **BAUDOT** invokes the Baudot character code set, **PTTC6I** invokes PTTC/6, etc.

Pragmatics

For special applications a programmer can define and invoke non-standard character codes by:

a. defining a translation table in a ⟨*translatetable definition*⟩ ;

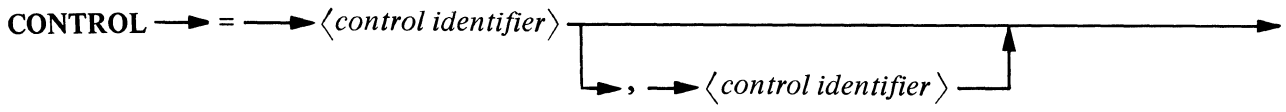
b. specifying **BINARY** or **EBCDIC** in the ⟨*terminal code statement*⟩ ; and

c. invoking the translation in a ⟨*control definition*⟩ or ⟨*request definition*⟩ by means of the appropriate option of the ⟨*assignment statement*⟩.

Refer to the ⟨*translatetable definition*⟩ in this chapter for more information.

TERMINAL CONTROL STATEMENT

Syntax



Examples

```
CONTROL = CONTENTION.
CONTROL = PRIMARYCONTROL,
          AUXILIARYCONTROL.
```

Semantics

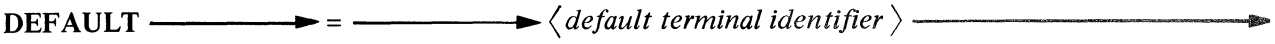
The ⟨*terminal control statement*⟩ specifies the ⟨*control definition*⟩*(s)* responsible for allocation of the logical line(s) to which a terminal type is associated. This attribute must be defined for all ⟨*terminal definitions*⟩.

Terminal types that do not utilize full duplex, reverse channel, or voice response features require that only one ⟨*control identifier*⟩ be named.

Terminal types that utilize full duplex, reverse channel, or voice response features (i.e., **DUPLEX = TRUE**) may optionally specify a second ⟨*control identifier*⟩. The first ⟨*control identifier*⟩ names the ⟨*control definition*⟩ for the primary line, and the second ⟨*control identifier*⟩ names the ⟨*control definition*⟩ for the auxiliary line. If only one ⟨*control identifier*⟩ is specified, it is assumed to be the ⟨*control definition*⟩ for the primary line, and the default equivalent of an ⟨*idle statement*⟩ is used for auxiliary line control.

TERMINAL DEFAULT STATEMENT

Syntax



Example

```
DEFAULT = TTYDFLT.
```

Semantics

The ⟨*terminal default statement*⟩ allows the programmer to specify the ⟨*default terminal identifier*⟩ of a set of default terminal attributes previously defined to be used for a ⟨*terminal definition*⟩ whose description is incomplete. It is advantageous to group common attributes under a Default ⟨*terminal definition*⟩ and list the remaining attributes under each individual ⟨*terminal definition*⟩. The compiler will then refer to the Default ⟨*terminal definition*⟩ to complete the ⟨*terminal definition*⟩. The ⟨*terminal default statement*⟩ is not required to appear in ⟨*terminal definitions*⟩; however, a ⟨*terminal definition*⟩ must define all required attributes if a ⟨*terminal default statement*⟩ does not appear.

The ⟨*terminal default statement*⟩ can appear in a ⟨*terminal definition*⟩ or a Default ⟨*terminal definition*⟩.

Supplementary Example

The following example illustrates how ⟨*terminal default statement*⟩*s* may be "nested" to combine the attributes of one or more Default ⟨*terminal definition*⟩*s*.

The effect of referencing GENERALDEFAULT within the Default ⟨*terminal definition*⟩ TTYDEFAULT is that the attributes associated with TTYDEFAULT are equivalent to all attributes as defined by GENERALDEFAULT plus the attributes explicitly defined in TTYDEFAULT.

If a ⟨*terminal definition*⟩ or Default ⟨*terminal definition*⟩ references a Default ⟨*terminal definition*⟩, the compiler does not compare the two definitions for contradictory statements. If contradictory statements exist within the two definitions, the last value defined for the attribute takes precedence. In the example, TTY2 defines the value of the **PAGE** attribute as 66, and the Default ⟨*terminal definition*⟩ that TTY2 references defines the value of the **PAGE** attribute as 0. The compiler uses 66 as the value of the **PAGE** attribute for TTY2.

```
TERMINAL DEFAULT GENERALDEFAULT:

    TURNAROUND    = 0.
    ICTDELAY      = 0.
    TRANSMISSION  = 0.
    ADDRESS       = 0.
    PAGE          = 0.
    BUFFER        = 0.

TERMINAL DEFAULT TTYDEFAULT:

    DEFAULT       = GENERALDEFAULT.
    BLOCK         = FALSE.
    SCREEN        = FALSE.
    PARITY        = NULL.
    SYNCS         = FALSE.
    TIMEOUT       = 3 SEC.
    MAXINPUT      = 72.
    WIDTH         = 72.
    ADAPTER       = 4.
    CODE          = ASC67.

TERMINAL TTY1:

    DEFAULT       = TTYDEFAULT.
    DUPLEX        = FALSE.
    WRU           = ENQ.
    END           = ETX (DYNAMIC).
    BACKSPACE     = BS (DYNAMIC).
    CONTROL       = CONTEND.
    REQUEST       = WRITETTY: TRANSMIT, READTTY: RECEIVE.

TERMINAL TTY2:

    DEFAULT       = TTYDEFAULT.
    DUPLEX        = FALSE.
    WRU           = 4"98".
    BACKSPACE     = 4"97".
    CONTROL       = SPECIALCNTRL.
    REQUEST [1]   = READER: RECEIVE, WRITER: TRANSMIT.
    REQUEST [2]   = READPPT: RECEIVE, WRITEPPT: TRANSMIT.
    PAGE          = 66.
```

Default ⟨*terminal definition*⟩

⟨*terminal default statement*⟩ references above Default ⟨*terminal definition*⟩*s*.

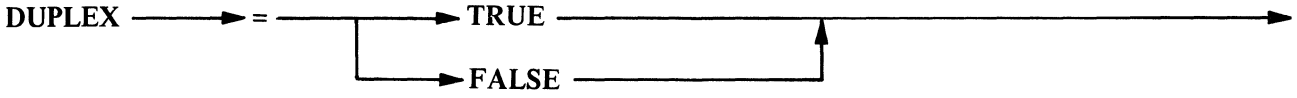
Default ⟨*terminal definition*⟩

⟨*terminal default statement*⟩ references above Default ⟨*terminal definition*⟩*s*.

⟨*terminal default statement*⟩ references above Default ⟨*terminal definition*⟩*s*.

TERMINAL DUPLEX STATEMENT

Syntax

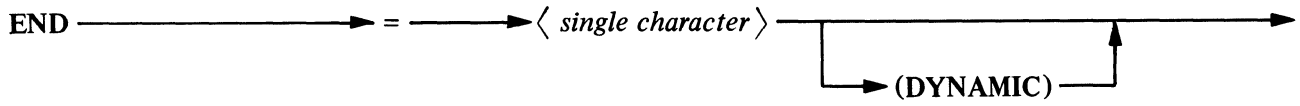


Semantics

The ⟨*terminal duplex statement*⟩ defines whether or not (**TRUE** or **FALSE**, respectively) the terminal type utilizes full duplex, reverse channel, or voice response features. If **DUPLEX = TRUE**, then the ⟨*line definition*⟩ for any line that has this terminal type assigned must contain the ⟨*line type statement*⟩ constructs that specify full duplex. This attribute must be defined for each ⟨*terminal definition*⟩.

TERMINAL END CHARACTER STATEMENT

Syntax



Examples

```
END = 4"0D".
END = "&" (DYNAMIC).
```

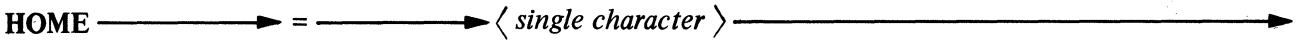
Semantics

The ⟨*terminal end character statement*⟩ defines the "end" character of the terminal type (i.e., the character that the terminal type would transmit to indicate an end-of-text). If defined, the "end" character can be recognized by the DCP when **RECEIVE**d (in a ⟨*receive statement*⟩), and any action to be taken can be specified by the programmer (using the **END** syntax).

(DYNAMIC) indicates that the Message Control System of a station referencing the ⟨*terminal definition*⟩ is allowed to change the character for the station by means of a SET CHARACTERS (TYPE=39) DCWRITE.

TERMINAL HOME CHARACTER STATEMENT

Syntax



Example

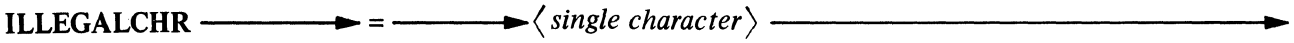
```
HOME = 4"0C".
```

Semantics

This statement is implemented for program documentation purposes only. It provides a means of documenting the home character of the terminal type. The documentation of this character in a ⟨*terminal definition*⟩ is optional.

TERMINAL ILLEGAL CHARACTER STATEMENT

Syntax



Example

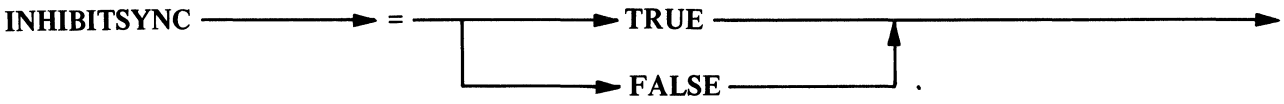
```
ILLEGALCHR = 4"FF".
```

Semantics

The ⟨*terminal illegal character statement*⟩ is implemented for documentation purposes only. The documentation of this character is not required in a ⟨*terminal definition*⟩.

TERMINAL INHIBITSYNC STATEMENT

Syntax



Semantics

The ⟨*terminal inhibitsync statement*⟩ affects only terminal types that specify any of the ⟨*communication type number*⟩*s* 17 through 27 in its ⟨*terminal adapter statement*⟩. This statement has no affect upon, and need not be defined for, terminal types that do not specify any of those ⟨*communication type number*⟩*s*.

If **INHIBITSYNC = FALSE**, then the following occurs during a synchronous transmission. The transmission begins with the transmission of four sync characters by the adapter cluster. As the fourth sync character is being transmitted, the first character of the message is requested from the DCP. The DCP should respond to this request by supplying the first character of the transmission. As each supplied character is transmitted, the adapter cluster requests another character. If the DCP is unable to respond in time to the request, the adapter cluster transmits a sync character; this process is called "sync filling." Sync filling is repeated as necessary until the DCP responds with another character or the DCP directs the adapter cluster to "finish transmit" for the line.

When **INHIBITSYNC = FALSE** during a synchronous reception, the following occurs. At the beginning of the reception, bit patterns from the line are examined by the adapter cluster and the bits discarded until a sync character is recognized. The recognition of a sync character establishes that the next bit to be received by the adapter cluster is the first bit of the next character. The sync character is discarded, instead of being made available to the DCP. All characters in the transmission that are not sync characters are made available to the DCP. The DCP may then fetch these characters. Any sync characters received in the transmission are discarded.

If **INHIBITSYNC = TRUE** , then the following occurs during a synchronous transmission. All actions occur that would occur if **INHIBITSYNC = FALSE**. In addition, if a sync fill is required, a "sync fill interrupt" occurs so that the DCP can determine when one or more undesired sync characters have been inserted into the transmission. System software responds to the interrupt by executing a **TERMINATE ERROR**. The controlling MCS is notified of all such situations so that corrective action (MAKE LINE READY (TYPE = 96) DCWRITE, for example) can be taken.

When **INHIBITSYNC = TRUE** during a synchronous reception, the following occurs. At the beginning of the reception, bit patterns from the line are examined by the adapter cluster and the bits discarded until a sync character is recognized. The recognized sync character is discarded, as is the next character if it is also a sync character. Thereafter, all subsequent characters (sync characters or otherwise) are made available to the DCP as data.

The reserved word **SYNCS** is a synonym for **INHIBITSYNC**.

TERMINAL INTER-CHARACTER DELAY STATEMENT

Syntax

Examples

```
ICTDELAY = 0.
ICTDELAY = 200 MILLI.
```

Semantics

The ⟨*terminal inter-character delay statement*⟩ provides the user a means to insert a timed delay between each character transmitted to the terminal type. The ⟨*delay time*⟩ specified defines the interval of ⟨*time*⟩ between the transmission of the start of one character to the start of the next character. If the time specified is less than the time required to transmit a character, this statement has no effect. This attribute must be defined for all ⟨*terminal definition*⟩*s*.

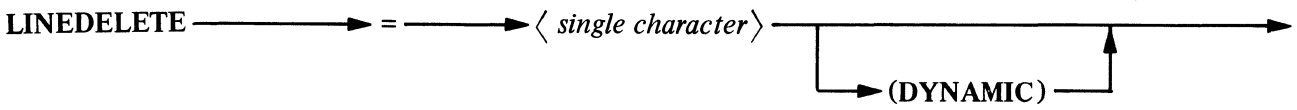
Supplementary Example

A Model 33 TELETYPE can receive characters at a maximum rate of one character every 100 milliseconds. If, for some reason, the programmer needs to insert a 100-millisecond delay between each character transmitted to the terminal, this can be done by specifying:

```
ICTDELAY = 200 MILLI.
```

TERMINAL LINEDELETE CHARACTER STATEMENT

Syntax



Examples

```
LINEDELETE = 4"07".
LINEDELETE = 4"A0".
```

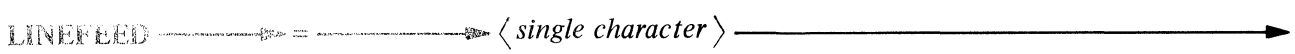
Semantics

The ⟨*terminal linedelete character statement*⟩ defines the linedelete character of the terminal type. If defined, the linedelete character can be recognized by the DCP when **RECEIVE**d (in a ⟨*receive statement*⟩), and any action to be taken can be specified by the programmer (using the **LINEDELETE** syntax).

(DYNAMIC) indicates that the Message Control System of a station referencing the ⟨*terminal definition*⟩ is allowed to change the character for the station by means of a SET CHARACTERS (TYPE=39) DCWRITE.

TERMINAL LINEFEED CHARACTER STATEMENT

Syntax



Example

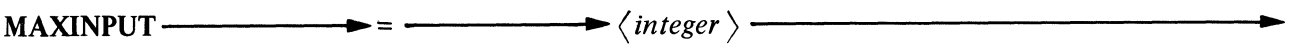
```
LINEFEED = 4"25".
```

Semantics

This statement is provided for program documentation only. It documents the linefeed character of the terminal type. The documentation of this character in a ⟨*terminal definition*⟩ is optional.

TERMINAL MAXINPUT STATEMENT

Syntax



Example

```
MAXINPUT = 72.
```

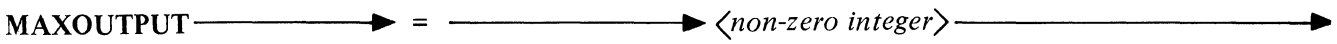
Semantics

The ⟨*terminal maxinput statement*⟩ defines the maximum size text, in characters, that a terminal is allowed to transmit in one message. This statement is synonymous with the ⟨*terminal buffer size statement*⟩. However, if the ⟨*terminal buffer size statement*⟩ is defined as non-**NULL**, it will override any ⟨*terminal maxinput statement*⟩.

If the ⟨*terminal maxinput statement*⟩ is omitted, then either a non-**NULL** ⟨*terminal buffer size statement*⟩ or a ⟨*terminal maxoutput statement*⟩ must be defined for the ⟨*terminal definition*⟩. The maximum terminal input will then default to the buffer size specification or the maximum terminal output specification in that order.

TERMINAL MAXOUTPUT STATEMENT

Syntax



Example

```
MAXOUTPUT = 1920.
```

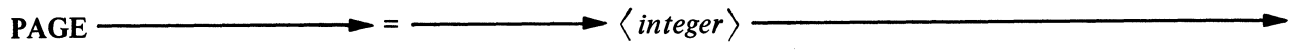
Semantics

The ⟨*terminal maxoutput statement*⟩ defines the maximum size text, in characters, that may be transmitted to a terminal in one message.

If the ⟨*terminal maxoutput statement*⟩ is omitted, then either a non-**NULL** ⟨*terminal buffer size statement*⟩ or a ⟨*terminal maxinput statement*⟩ must be defined for the ⟨*terminal definition*⟩. The maximum terminal output will then default to the value for the maximum terminal input.

TERMINAL PAGE STATEMENT

Syntax



Examples

```
PAGE = 0.
PAGE = 12.
```

Semantics

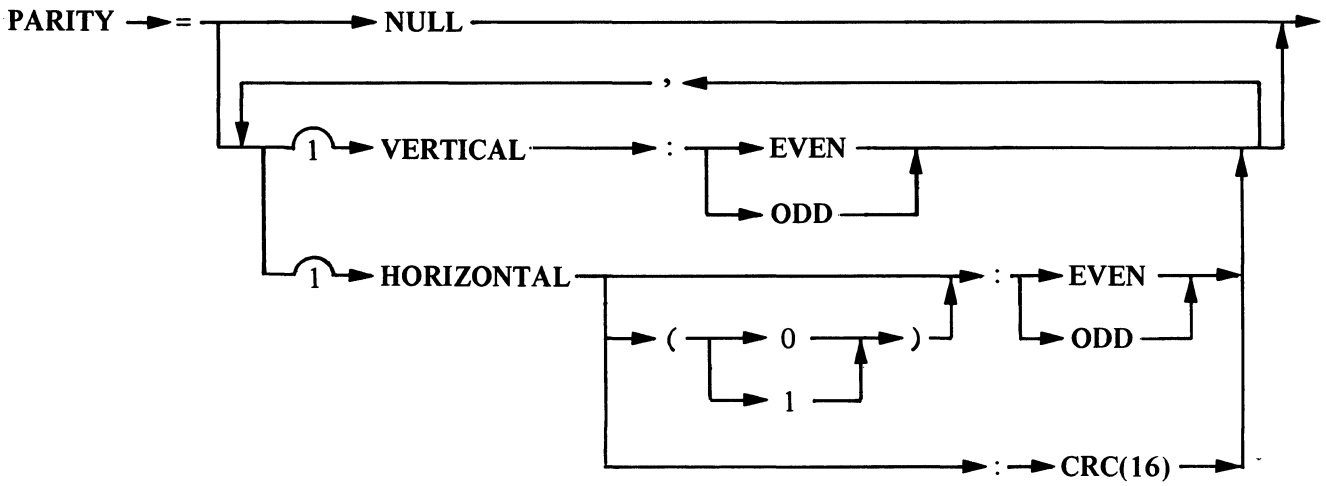
The ⟨*terminal page statement*⟩ defines the maximum number of output lines per page as restricted by the hardware of the terminal. There are, for example, devices that can only print/display a defined number of lines before some type of carriage/cursor control information must be supplied. If the terminal type being defined has no such restrictions, then

```
PAGE = 0.
```

should be specified, thus indicating that pagination is arbitrary. This attribute must be defined for all ⟨*terminal definition*⟩*s*.

TERMINAL PARITY STATEMENT

Syntax



Examples

```
PARITY = NULL.
PARITY = VERTICAL:ODD.
PARITY = HORIZONTAL:CRC(16).
PARITY = VERTICAL:ODD, HORIZONTAL(0):EVEN.
```

Semantics

The ⟨*terminal parity statement*⟩ defines the type of parity checking and generation to be performed by the DCP when communicating with the terminal type. If the form:

```
PARITY=NULL.
```

is used, parity is not checked or generated.

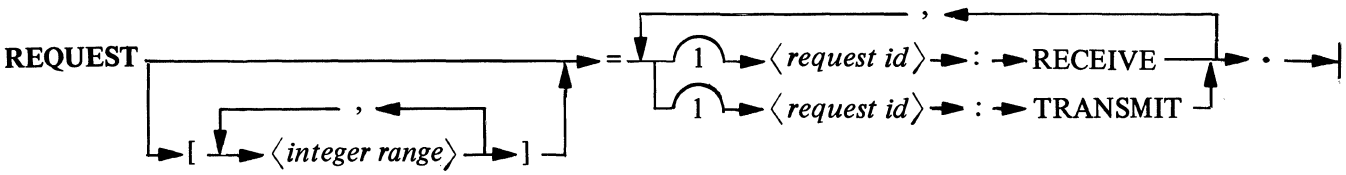
The **VERTICAL** option refers to the vertical parity bit of a character, and can be defined as **ODD** or **EVEN**.

The **HORIZONTAL** option specifies the type of horizontal parity. If horizontal parity is a Block Check Character, then **ODD** or **EVEN** must be specified. If horizontal parity is a Cyclic Redundancy Check, then **CRC(16)** must be specified.

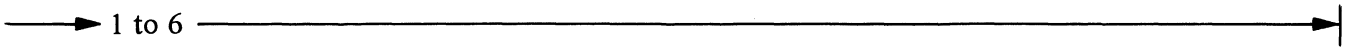
The **0** or **1** option defines the function of the vertical parity bit of the Block Check Character. If this bit is a parity bit for the Block Check Character, then this option must be omitted or defined as **0** (zero). If undefined, the option is assumed to be **0** (zero). If the bit is to be considered as a horizontal parity bit of all high-order bits in the message, then this option must be defined as **1**.

TERMINAL REQUEST STATEMENT

Syntax



⟨*integer range*⟩



Examples

```
REQUEST = READTTY:RECEIVE.
REQUEST[1, 5, 6] = WRITETTY:TRANSMIT,READTTY:RECEIVE.
REQUEST[3, 4] = WRITESCREEN:TRANSMIT,READSCREEN:RECEIVE.
```

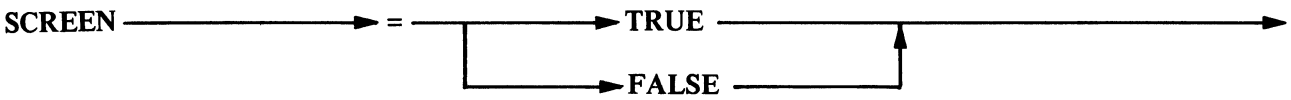
Semantics

The ⟨*terminal request statement*⟩ specifies a ⟨*request identifier*⟩ , or a pair of ⟨*request identifier*⟩s, that designates the ⟨*request definition*⟩ to handle input from (the **RECEIVE** option) and/or output to (the **TRANSMIT** option) the terminal type. The ⟨*request definition*⟩ that handles input is commonly referred to as the Receive Request, and the ⟨*request definition*⟩ that handles output is commonly referred to as the Transmit Request. This statement must appear in each ⟨*terminal definition*⟩, and cannot appear in a Default ⟨*terminal definition*⟩.

The {⟨*integer*⟩ value of 1 through 6} allows the specification of up to six pairs of Transmit and Receive Requests for the same device. Normally, these Request pairs differ for some application-dependent reasons. Only one pair of ⟨*request definition*⟩ s can be the controlling ⟨*request definition*⟩ at any instant of time. The ⟨*request definition*⟩ in control of the terminal type immediately after DCP initialization has an ⟨*application number*⟩ of 1x, which retains control until the Message Control System (MCS) of a station associated with the terminal type executes a SET APPLICATION NUMBER (TYPE=38) DCWRITE. The correct value of ⟨*application number*⟩ is always available in the station byte variable **APPLICATION**.

TERMINAL SCREEN STATEMENT

Syntax



Example

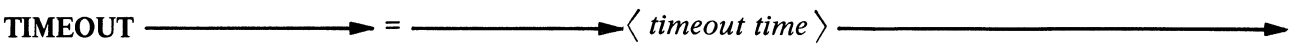
```
SCREEN = TRUE.
```

Semantics

The ⟨*terminal screen statement*⟩ defines whether or not (**TRUE** or **FALSE**, respectively) the terminal type is a screen (i.e., CRT) device. This attribute must be defined in each ⟨*terminal definition*⟩.

TERMINAL TIMEOUT STATEMENT

Syntax



Example

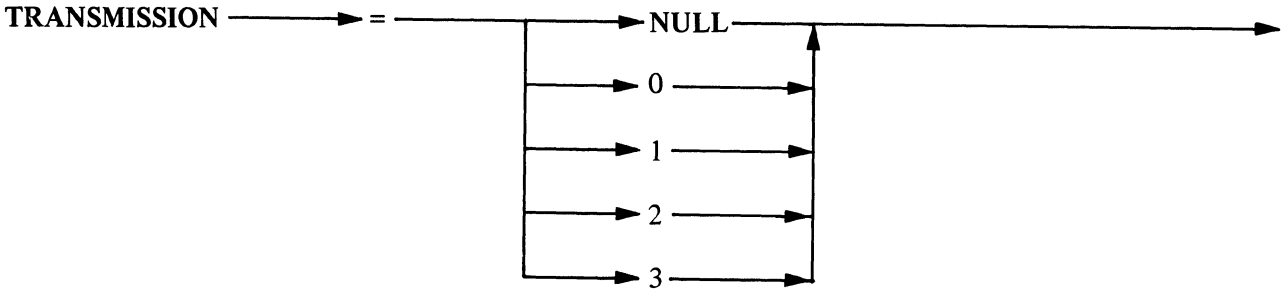
```
TIMEOUT = 3 SEC.
```

Semantics

The ⟨*terminal timeout statement*⟩ defines the interval of ⟨*time*⟩ that the adapter cluster should wait from the receipt of one character to the start of the next (in a ⟨*receive statement*⟩) before assuming that the terminal has "timed out." The action taken upon a timeout condition can be specified in a ⟨*receive statement*⟩ by means of the **TIMEOUT** syntax.

TERMINAL TRANSMISSION NUMBER LENGTH STATEMENT

Syntax



Example

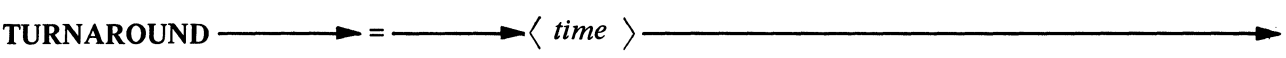
```
TRANSMISSION = 2.
```

Semantics

The ⟨*terminal transmission number length statement*⟩ defines the number of characters that the terminal transmits and receives as the message transmission number. The **0** and **NULL** options are semantically equivalent and specify that no transmission number is used. A non-**NULL** transmission number length must be specified if a ⟨*control definition*⟩ or ⟨*request definition*⟩ that references the item **TRAN** is defined for the terminal type. If a **NULL** transmission number **LENGTH** is specified, constructs such as **RECEIVE TRAN**, **TRANSMIT TRAN** and **INCREMENT TRAN** will be ignored.

TERMINAL TURNAROUND STATEMENT

Syntax



Examples

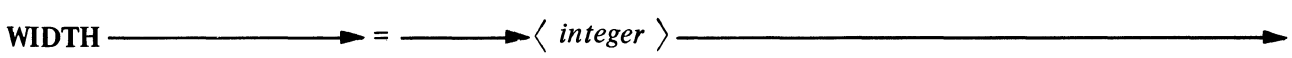
```
TURNAROUND = 0.
TURNAROUND = 200 MILLI.
```

Semantics

The ⟨*terminal turnaround statement*⟩ defines the time required for the terminal to shift from transmitting data to receiving data. The ⟨*time*⟩ defined is a parameter of a compiler algorithm for calculating the initiate transmit delay. Refer to the semantics of the ⟨*control definition*⟩ or ⟨*request definition*⟩ ⟨*initiate statement*⟩ in this chapter for more information. This attribute must be defined for each ⟨*terminal definition*⟩.

TERMINAL WIDTH STATEMENT

Syntax



Example

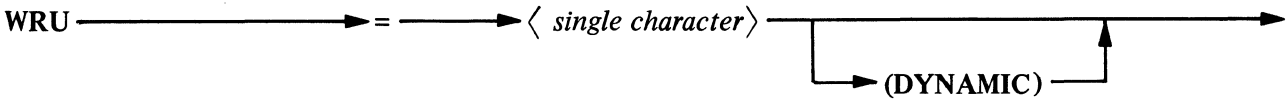
```
WIDTH = 80.
```

Semantics

The ⟨*terminal width statement*⟩ defines the width, in characters, of a display line of output on the terminal type. The ⟨*integer*⟩ must be greater than 0 and less than 256; additionally, the value of the ⟨*integer*⟩ must be less than or equal to the size defined in the ⟨*terminal buffer size statement*⟩, if present. It is not required that the ⟨*terminal width statement*⟩ appear in a ⟨*terminal definition*⟩. If the ⟨*terminal width statement*⟩ is not defined in the ⟨*terminal definition*⟩, then the buffer size value is substituted for this value, if present; otherwise, the value of **MAXINPUT** is substituted by default.

TERMINAL WRU CHARACTER STATEMENT

Syntax



Examples

```
WRU = 4"2D" (DYNAMIC).
WRU = "?".
```

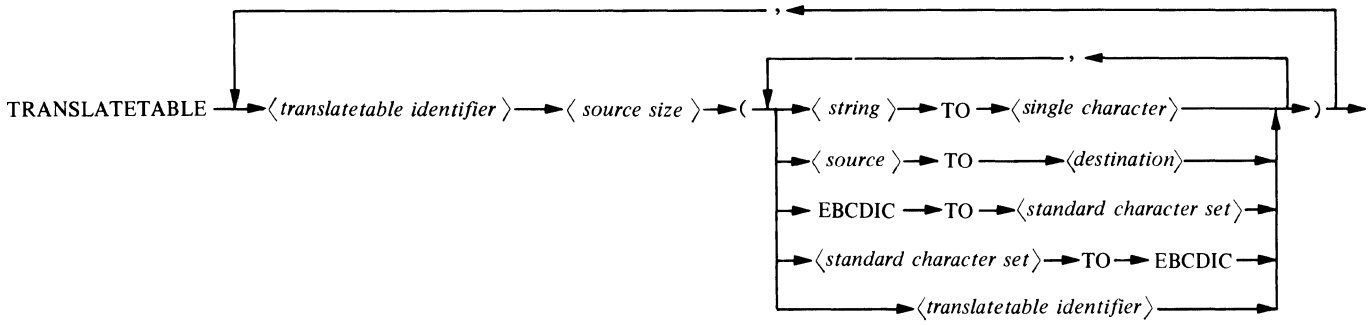
Semantics

The ⟨*terminal WRU character statement*⟩ defines the **WRU** character for the terminal type (i.e., the character the terminal type would transmit to request a response from the DCP). If defined, the **WRU** character can be recognized by the DCP when **RECEIVE**d (in a ⟨*receive statement*⟩), and any action to be taken can be specified by the programmer (using the **WRU** syntax).

(DYNAMIC) indicates that the Message Control System of a station referencing the ⟨*terminal definition*⟩ is allowed to change the character for the station by means of the SET CHARACTERS (TYPE=39) DCWRITE.

TRANSLATETABLE DEFINITION

Syntax



Examples

```
TRANSLATETABLE   ATABLE   8("STRING" TO "X").
TRANSLATETABLE   BTABLE   8(4"000102" TO 4"AABBCC").
TRANSLATETABLE   CTABLE   7(4"02820B" TO 4"AACCDD").
TRANSLATETABLE   DTABLE   7(4"00" TO 4"AA"),
                 ETABLE   8(DTABLE, 4"01" TO 4"BB").
TRANSLATETABLE   FTABLE   8(4"00" TO 4"AA"),
                 GTABLE   8(EBCDIC TO BCL,FTABLE,
                            4"01" TO 4"BB").
TRANSLATETABLE   TRANID   7("1" TO 4"01",
                            "2" TO 4"02".
                            "34" TO 4"0304",
                            4"F5F6" TO 4"0607").
```

Semantics

The ⟨*translatetable definition*⟩ allows the definition of tables that may be used in ⟨*control definition*⟩*s* or ⟨*request definition*⟩*s* to translate characters of one character set to those of another character set.

Translation tables need to be defined in an NDL program only if non-standard character sets must be dealt with in the Data Communications System. Terminals that transmit and receive a standard character set do not require a translation table definition; instead, the character set is merely named in the ⟨*terminal code statement*⟩ of the ⟨*terminal definition*⟩. The character sets that do not require a ⟨*translatetable definition*⟩ are ASCII, BAUDOT, BCD, BCL, EBCDIC, and PTTC/6.

The ⟨*translatetable identifier*⟩ that follows the key word **TRANSLATETABLE** names the translation table, and must be in the syntactic form of an ⟨*identifier*⟩.

⟨*source size*⟩ defines the character size, in bits, of characters to be translated. ⟨*source size*⟩ must be an ⟨*integer*⟩ greater than 0 and less than 9.

TRANSLATION TABLE STRUCTURE

Each element of the translation table consists of eight bits. If N represents the ⟨*source size*⟩, then the size of the table is 2 raised to the Nth power. The elements of the table are selected by an index that ranges from 0 through 2 to the Nth power minus 1.

At execution time, translation is done in the following manner. The binary weight of the low-order N bits of the character to be translated is used as an index into the specified translation table. The element of the table thus indexed is the translated result.

INSERTING DATA INTO THE TRANSLATION TABLE

Every translation table has a default base in which each element in the table is 0 (all bits off). Data can be placed into the translation table by various specifications within the parentheses. If more than one specification appears for a given translation table, each succeeding specification overrides, within its scope, previous specifications.

⟨*string*⟩ **TO** ⟨*single character*⟩

This form inserts data into the translation table in the following manner. Each eight-bit character in the ⟨*string*⟩ is examined from left to right. If a character in the ⟨*string*⟩ is numerically greater than the size of the table, no entry is placed in the translation table; otherwise, the ⟨*single character*⟩ is stored in the element of the table whose index is the binary weight of the N low-order bits of the ⟨*string*⟩ character (where N is the ⟨*source size*⟩ specified).

⟨*source*⟩ **TO** ⟨*destination*⟩

⟨*source*⟩ and ⟨*destination*⟩ must be ⟨*string*⟩*s* of equal length. This form of specification inserts data into the translation table in the following manner. Translation is based upon corresponding characters in ⟨*source*⟩ and ⟨*destination*⟩, starting from left and proceeding to right. The first character of ⟨*source*⟩ corresponds to the first character of ⟨*destination*⟩, the second character of ⟨*source*⟩ corresponds to the second character of ⟨*destination*⟩, etc. If a character in ⟨*source*⟩ is numerically greater than the size of the table, then no entry is placed in the translation table; otherwise, the corresponding character in ⟨*destination*⟩ is stored in the element of the table whose index is the binary weight of the N low-order bits of the corresponding character in ⟨*source*⟩.

⟨*standard character set*⟩ **TO EBCDIC** and **EBCDIC TO** ⟨*standard character set*⟩

This form specifies a standard system software translation table from the NDL compiler that is to be copied into the translation table. The ⟨*standard character set*⟩*s* that may be specified are EBCDIC, ASC67, and BCL. These forms provide a way of obtaining a legitimate base upon which additional specifications can be made.

⟨*translatetable identifier*⟩

This form of specification indicates that the contents of a previously defined translation table is to be copied into the translation table. The ⟨*translatetable identifier*⟩ must be the ⟨*identifier*⟩ of a previously defined translation table. This form provides a means of obtaining a legitimate base upon which additional specifications can be made.

Pragmatics

Those tables, and only those tables, that are used by a DCP reside in the local memory of that DCP (unless a DCP does not have local memory, in which case they reside in main system memory). Memory for translation tables is allocated in blocks of 256 words, regardless of the space required for those tables. Tables are densely packed and all elements are used before another block of 256 words is allocated. Unless consideration is given to the translation requirements of devices in the data communications system while in the planning and programming stages, translation tables can be very costly in terms of local memory. Although it is beyond the scope of this manual to describe the planning of a data communications system, this fact should not escape the NDL programmer.

Supplementary Examples

Example 1

TRANSLATETABLE ATABLE 8("STRING" TO "X").

Character to be Translated	Result
"S"	"X"
"T"	"X"
"R"	"X"
"I"	"X"
"N"	"X"
"G"	"X"

ATABLE is a translation table containing 256 elements. The ⟨*source size*⟩, 8 in this example, determines the table size. All characters from ⟨*source*⟩ are translated to the ⟨*single character*⟩.

Example 2

TRANSLATETABLE BTABLE 8(4"000102" TO 4"AABBCC").

Character to be Translated	Result
4"00"	4"AA"
4"01"	4"BB"
4"02"	4"CC"
4"03"	4"00"

BTABLE contains 256 elements. Characters from ⟨*source*⟩, 4"000102", are translated to the corresponding characters in the ⟨*destination*⟩, 4"AABBCC". The character 4"03" is translated to 4"00" because there is no specification in ⟨*source*⟩ for 4"03".

Example 3

TRANSLATETABLE CTABLE 7(4"02820B" TO 4"AACCDD").

Character to be Translated	Result
4"01"	4"00"
4"02"	4"AA"
4"82"	4"AA"
4"0B"	4"DD"

In this example, the translation table CTABLE contains 128 elements. The character 4"01" is translated to a 4"00" character, because 4"01" is unspecified in the ⟨*source*⟩. The character 4"82" is translated to the character 4"AA" because only the low-order seven bits of 4"82" are used to index the translation table.

Example 4

```
TRANSLATETABLE  DTABLE 8(4"00" TO 4"AA").
                ETABLE 8(DTABLE, 4"01" TO 4"BB").
```

The above ⟨*translatetable definition*⟩ defines two translation tables: DTABLE and ETABLE. All elements in DTABLE contain 4"00", except the element indexed by the character 4"00"; that element contains 4"AA". ETABLE specifies DTABLE as a base, and then modifies that base with a subsequent specification.

Example 5

```
TRANSLATETABLE  FTABLE 8(4"00" TO 4"AA").
                GTABLE 8(EBCDIC TO BCL, FTABLE, 4"01" TO 4"BB").
```

GTABLE is defined to contain 256 elements, and specifies the standard EBCDIC-to-BCL translation table upon which subsequent specifications modify. FTABLE also contains 256 elements and appears as a specification in GTABLE. Since each succeeding specification overrides within its scope any previous specification, FTABLE in effect overlays all elements. The result is the same as if only the following had appeared:

```
TRANSLATETABLE  FTABLE 8(4"00" TO 4"AA"),
                GTABLE 8(FTABLE, 4"01" TO 4"BB").
```

The above example points out that any table appearing as a specification indicates all elements of that table, not just those elements explicitly defined. The example is not intended to illustrate an acceptable programming practice.

CHAPTER 6

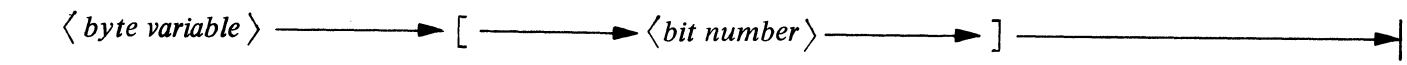
VARIABLES

GENERAL

The NDL compiler does not allow a programmer to declare and use program variables, as do other language compilers such as ALGOL, PL/I, and COBOL. Instead, the NDL programmer can use only predefined program variables.

The ⟨*bit variable*⟩*s* and ⟨*byte variable*⟩*s* are the two types of variables the programmer can use. The ⟨*bit variable*⟩*s* are one-bit variables that can only assume logical values (i.e., TRUE or FALSE). The ⟨*byte variable*⟩*s* are all eight-bit variables, and can assume integer values from 0 through 255, except for the **IR** variable, which is a 10-bit variable. The **IR** variable is included as a ⟨*byte variable*⟩ as a matter of convenience.

Individual bits of a ⟨*byte variable*⟩ can be referenced and used like a ⟨*bit variable*⟩, if referenced in the form illustrated below.



where ⟨*bit number*⟩ is an ⟨*integer*⟩ not greater than the number of bits contained in the variable minus 1.

For example, bit 5 of **IR** is referenced as **IR[5]**.

FUNCTION OF VARIABLES

Functionally, variables fall into one of three general categories:

a. Variables that are available to the programmer for general information storage.

b. Variables that can be used for system/station communication.

c. Variables that contain control information.

General information variables can be used within their scope by the programmer for data storage, calculations, etc. Additionally, some variables in this category could (by convention) be used as communication paths between ⟨*request definition*⟩*s* executing on a given line. (The use of a given variable for this application is restricted by the scope of that variable.)

Variables whose intended function is communication to and from the main system and stations are generally contained in the message header of a message sent to the main system from a station, or sent to the station from the main system. Messages from the main system to a station are originated either by the MCS or by an application program (via the I/O Intrinsics).

The format of message variables within a message header is described in detail in the B 5000/B 6000/B 7000 Series DCALGOL Reference Manual. Generally, message variables are contained in five fields of the message header:

a. Message Toggles (word [1].[39:8])

b. Message Tallys (word [3].[23:24])

c. Message Error Flags (word [1] . [23:24])

d. Variant "Carriage Control" (word [0] . [39:16])

e. Message Retry Count (word [2] . [47:8])

Message Toggles and Message Tallys provide storage area in the header for some of the station general information variables. The meaning of values stored in these fields must be established by mutual convention between the MCS writer and the NDL programmer.

Message Error Flags are used for the station to communicate to an MCS that some exceptional event has occurred in a ⟨*request definition*⟩ or ⟨*control definition*⟩. These variables reference bits in the message header of "result" messages returned to the MCS as a result of execution of a ⟨*terminate statement*⟩.

Carriage Control is valid for Transmit Requests, and provides information regarding the kind of carriage control to be performed by a Transmit Request. These variables reference bits or bytes in the message header of WRITE (TYPE=33) DCWRITE messages.

The Message Retry Count is described under **RETRY** in this chapter.

Variables whose function is to contain control information are used by both the DCP operating system and the programmer. Generally, these variables provide information to control the logic paths of ⟨*control definition*⟩*s*, ⟨*request definition*⟩*s*, and the DCP operating system.

SCOPE OF VARIABLES

The scope of the variables in NDL is described as being:

a. Station-oriented.

b. Line-oriented.

c. Global.

Station-oriented variables exist for each station in the network. **TALLY [0]** is an example of a station-oriented ⟨*byte variable*⟩; thus, each station has its own **TALLY [0]**. The variables of a given station are visible to a line only while **STATION** is set to that station's "station index."

Line-oriented variables exist for each line on a DCP. The variables of a given line are visible to every station assigned to that line. **MAXSTATIONS** is an example of a line-oriented variable. Each line on a DCP has its own **MAXSTATIONS**, and every station assigned to a given line can access the **MAXSTATIONS** variable of that line.

A global variable is a variable that is visible to all stations on a DCP.

DESCRIPTION OF VARIABLES

The remainder of this chapter contains descriptions of each ⟨*bit variable*⟩ and ⟨*byte variable*⟩. The variables (listed in table 6–1) are described in alphabetical order. The name of the variable precedes a summary of the variable characteristics, followed by a detailed description of the variable.

The summary of the variable characteristics includes the places in the source program that the variable can be interrogated or altered, and the size, in bits, of the variable. In the summary, the word "Interrogate" indicates that the programmer can interrogate the variable. The word "Alter" indicates that the programmer can use the ⟨*bit variable*⟩ / ⟨*byte variable*⟩ as an ⟨*assignable bit variable*⟩ / ⟨*assignable byte variable*⟩. The corresponding letters "C", "T", and "R" in the summary refer to ⟨*control definition*⟩, Transmit Request, and Receive Request, respectively. The last item to appear in the summary is the size, in bits, of the variable. If no size is defined, then the size of the variable is one bit.

For example, the summary:

EXAMPLE1
Interrogate, CTR, 8

can be expanded as follows:

EXAMPLE1 is an 8-bit variable. It can be interrogated in a ⟨*control definition*⟩, Transmit Request, or Receive Request.

The summary:

EXAMPLE2
Interrogate/Alter, CTR/TR

can be expanded as follows:

EXAMPLE2 is a ⟨*bit variable*⟩. It can be interrogated in a ⟨*control definition*⟩, Transmit Request, or Receive Request. Additionally, **EXAMPLE2** can be altered (i.e., appear as an ⟨*assignable bit variable*⟩) in a Transmit Request or Receive Request, but not in a ⟨*control definition*⟩.

Table 6–1 contains the summaries of each variable for quick reference.

Table 6–1. Table of Variables

NAME	SIZE (in bits)	INTERROGATE	ALTER
ADAPTER (ADAPTOR)	8	CTR	CTR
ADDERR	8	TR	TR
ADDRESS		CTR	CTR
AI	8	CTR	CTR
APPLICATION	8	CTR	--
AUX (LINE (BUSY))		CTR	CTR
AUX (LINE (QUEUED))		CTR	CTR
AUX (LINE (TALLY[tally number]))	8	CTR	CTR
AUX (LINE (TOG[toggle number]))		CTR	CTR
BACKSPACE (BKSP)	8	CTR	--
BCC	8	CTR	CTR
BCCERR		TR	TR
BLOCK		T	--
BLOCKED		T	--
BREAK [RECEIVE]		TR	TR
BREAK [TRANSMIT]		TR	TR
BUFOVFL		TR	TR
CARRIAGE		T	--
CARRIER		CTR	--
CHARACTER (CHAR)	8	CTR	CTR
CONTROL	8	CTR	--
CONTROLFLAG		TR	TR
CRC		CTR	CTR
CRC [{*0 or 1*}]	8	CTR	CTR
CRCERR		TR	TR
DISCONNECT		TR	TR
END	8	CTR	--
ENDOFBUFFER		TR	TR
FORMATERR		TR	TR
INHIBITSYNC		CTR	CTR
IR	10	CTR	--
LINE (BUSY)		CTR	CTR
LINE (QUEUED)		CTR	CTR
LINE (TALLY[tally number])	8	CTR	CTR
LINE (TOG[toggle number])		CTR	CTR
LINEDELETE (LD)	8	CTR	--
LINEFEED		T	--
LOSSOFCARRIER		TR	TR
MAXSTATIONS	8	CTR	--
NAKFLAG		TR	TR

Table 6–1. Table of Variables (Cont)

NAME	SIZE (in bits)	INTERROGATE	ALTER
NAKONSELECT		TR	TR
NOSPACE		CTR	------
PAGE		T	------
PAPERMOTION		T	------
PARITY		TR	TR
RETRY	8	CTR	CTR
SEQERR		TR	TR
SEQUENCE		CTR	CTR
SKIP		T	------
SKIPCONTROL	8	T	------
SPACE		T	------
STATION	8	CTR	C
STATION (ENABLED)		CTR	------
STATION (FREQUENCY)	8	CTR	------
STATION (QUEUED)		CTR	------
STATION (READY)		CTR	------
STATION (TALLY)	8	CTR	CTR
STATION (VALID)		CTR	------
STOPBIT		TR	TR
SYNCS		CTR	CTR
SYSTEM DUMPING		CTR	------
TALLY [⟨*tally number*⟩]	8	CTR	CTR
TIMEOUT		TR	TR
TOG [⟨*toggle number*⟩]		CTR	CTR
TOGGLES [⟨*toggles number*⟩]	8	CTR	CTR
TOGS	8	CTR	CTR
TRAN	8	CTR	CTR
TRANERR		TR	TR
TRANSMITMODE	1	CTR	------
WRU	8	CTR	------
WRUFLAG		TR	TR

ADAPTER (ADAPTOR)
Interrogate/Alter, CTR/CTR, 8

ADAPTER is a ⟨*byte variable*⟩ which references the ⟨*communication type number*⟩ as an integer in the range of 1 through 15. It is meaningful on asynchronous lines only.

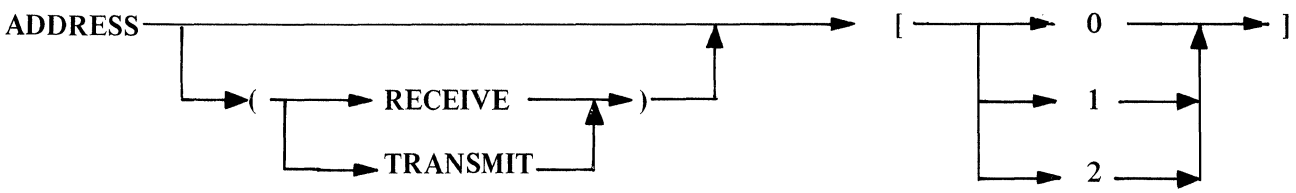
ADAPTER may be assigned values, thereby dynamically changing the band rate and, optionally, the character size of the line adapter. All values assigned to **ADAPTER** will be stored modulo 16. The new adapter type will remain in effect until the data communications subsystem is reinitialized or until a reconfiguration request is performed.

On synchronous lines, the value of **ADAPTER** is always zero, and any attempt to change the value is ignored.

ADDERR
Interrogate/Alter, TR/TR

ADDERR references bit 8 in the Error Flag Field of a message header, and normally indicates that an address character error has occurred while executing a ⟨*receive statement*⟩. Refer to the **ADDRESS** option of the ⟨*receive statement*⟩.

ADDRESS
Interrogate/Alter, CTR/CTR



Example:

ADDRESS (TRANSMIT) [1] = 4"05"

These bytes allow access to a stations **ADDRESS** characters.

They are grouped as two sets of three bytes each, one set for transmit and one for receive. Within each group, the three bytes are numbered from zero. Byte zero represents the least significant byte. The "**RECEIVE**" or "**TRANSMIT**" part may be omitted, in which case the receive bytes will be selected in a receive request or control definition and the transmit bytes in a transmit request.

Within each station table, the address bytes are stored from left to right as follows:

ADDRESS(TRANSMIT) [2]
ADDRESS(TRANSMIT) [1]
ADDRESS(TRANSMIT) [0]
ADDRESS(RECEIVE) [2]
ADDRESS(RECEIVE) [1]
ADDRESS(RECEIVE) [0]

Each address is stored in **EBCDIC**, right-justified with high-order binary zeroes within their respective half-word. Thus, a station with a two-character address of "5A" would present the following values when accessed:

ADDRESS [0] = "A"
ADDRESS [1] = "5"
ADDRESS [2] = 4"00"

AI
Interrogate/Alter, CTR/CTR, 8

This variable addresses a volatile register and should not be used for data storage. Its main purpose is to allow access to the untranslated byte just received rather than to the translated byte in **CHARACTER**, particularly when executing the ⟨*sum statement*⟩.

APPLICATION
Interrogate, CTR, 8

This read-only statement-oriented byte variable contains the value supplied in the most recent SET APPLICATION NUMBER DCWRITE (type 38). It is limited to integers from 1 to 6. When the station is initialized, APPLICATION is given a value of 1.

Example

IF APPLICATION=6 THEN . . .

AUX(LINE(BUSY))
Interrogate/Alter, CTR/CTR

AUX(LINE(BUSY)) is used to allow or inhibit the interruption of the execution of a ⟨*control definition*⟩ or ⟨*request definition*⟩ on the auxiliary line of a full duplex line pair. If this bit is **TRUE**, it indicates to the DCP operating system that the line is engaged in functions that must not be interrupted. If **FALSE**, it indicates to the DCP operating system that the line can be interrupted to initiate another function. **AUX(LINE(BUSY))** is line-oriented, but may be altered only by the auxiliary line. Both the auxiliary and primary line may interrogate this bit.

A ⟨*control definition*⟩ or ⟨*request definition*⟩ will be interrupted when **AUX(LINE(BUSY))** is **FALSE** if the primary line executes a ⟨*fork statement*⟩. (Note that an interruption causes control to leave a ⟨*control definition*⟩ or ⟨*request definition*⟩, and that control is not returned to the point where the interruption occurred.) **AUX(LINE(BUSY))** is set **TRUE** by system software when:

a. The primary line executes a ⟨*fork statement*⟩ and **AUX(LINE(BUSY))** is **FALSE** or

b. The auxiliary line ⟨*control definition*⟩ is entered, or

c. The auxiliary line enters a Receive or Transmit Request.

If **AUX(LINE(BUSY))** is **TRUE** when the primary line executes a ⟨*fork statement*⟩, the ⟨*fork statement*⟩ will act as a no-op.

AUX(LINE(QUEUED))
Interrogate/Alter, CTR/CTR

This is a line-oriented ⟨*bit variable*⟩ that refers to the queued status of the auxiliary line of a full duplex line pair. The bit is set by the DCP operating system if and when an input message space is explicitly acquired by executing a ⟨*getspace statement*⟩ on the auxiliary line.

AUX(LINE(TALLY [tally number]))
Interrogate/Alter, CTR/CTR, 8

These are line-oriented ⟨*byte variable*⟩*s* for the auxiliary line of a full duplex line pair, and can be used for any purpose by the NDL programmer. They can be accessed by either the primary or auxiliary line at any time.

AUX(LINE(TOG [toggle number]))
Interrogate/Alter, CTR/CTR

These are line-oriented ⟨*bit variable*⟩*s* for the auxiliary line of a full duplex pair, and may be used for any purpose by the NDL programmer. They may be accessed by either the primary or auxiliary line at any time.

BACKSPACE (BKSP)
Interrogate, CTR, 8

This read-only station variable allows access to the **BACKSPACE** character for the station. This variable contains the value supplied in the most recent **SET CHARACTERS DCWRITE** (type=39) and is initialized to the value specified in the ⟨*terminal backspace character statement*⟩.

BCC
Interrogate/Alter, CTR/CTR, 8

BCC is used by system software for the purpose of accumulating a Block Check Character when a station ⟨*terminal definition*⟩ defines horizontal parity as **ODD** or **EVEN** in the ⟨*terminal parity statement*⟩.

Block Check Character accumulation is an automatic function, if appropriate, of the ⟨*receive statement*⟩ and ⟨*transmit statement*⟩. Block Check Character accumulation is based upon execusive-OR logic, that is, as characters are received or transmitted, they are exclusively OR-ed with the contents of **BCC**. It is the responsibility of the programmer to initialize **BCC** when appropriate. (Refer to the ⟨*initialize statement*⟩ under ⟨*request definition*⟩ or ⟨*control definition*⟩.)

If a station ⟨*terminal definition*⟩ does not define horizontal parity, **BCC** can be used as a temporary data storage area. It should be pointed out, however, that the value in **BCC** is destroyed by most constructs of the ⟨*terminate statement*⟩. Furthermore, since the intended purpose of **BCC** is to contain parity information, **BCC** and **CRC[0]** address the same data space. **BCC** cannot be used if a terminal uses Cyclic Redundancy Check.

When accumulating a Block Check Character, a convenient means to eliminate a specific character from the value accumulated in **BCC** is the ⟨*sum statement*⟩

BCCERR
Interrogate/Alter, TR/TR

BCCERR refers to bit 7 in the Error Flag Field of a result message, and conventionally indicates that a horizontal parity (**BCC**) error occurred while executing a ⟨*receive statement*⟩. Refer to the semantics of the **BCC** option of the ⟨*receive statement*⟩.

BLOCK
Interrogate/Alter, T/R

This bit references bit 29 in word zero of a message header. If **TRUE**, this bit indicates that more blocks (or messages) of a blocked transmission are to follow. Use of this bit implies a convention between the MCS and the NDL programmer for the purposes of providing blocked transmissions.

BLOCK is set **TRUE** implicitly as a result of execution of a **TERMINATE BLOCK** construct in a Receive Request.

BLOCKED
Interrogate/Alter, T/R

A synonym for **BLOCK**. Refer to **BLOCK**.

BREAK[RECEIVE]
Interrogate/Alter, TR/TR

This ⟨*bit variable*⟩ refers to bit 3 in the Error Flag Field of a message, and normally indicates that a break condition was sensed in a ⟨*receive statement*⟩. Refer to the semantics of the **BREAK** option of the ⟨*receive statement*⟩.

Note that if this bit is **TRUE** in a message to be returned to the MCS, the message is returned as a STATION EVENT (CLASS=1) message.

BREAK[TRANSMIT]
Interrogate/Alter, TR/TR

This ⟨*bit variable*⟩ refers to bit 5 in the Error Flag Field of a message, and normally indicates that a break condition was sensed while executing a ⟨*transmit statement*⟩. Refer to the semantics of the **BREAK** option of the ⟨*transmit statement*⟩.

Note that if this bit is **TRUE** in a message to be returned to the MCS, the message is returned as a STATION EVENT (CLASS=1) message. Refer to the B 5000/B 6000/B 7000 Series DCALGOL Language Reference Manual for more information regarding this message.

BUFOVFL
Interrogate/Alter, TR/TR

This ⟨*bit variable*⟩ refers to bit 2 in the Error Flag Field of a message, and normally indicates that a cluster buffer overflow condition occurred while executing a ⟨*receive statement*⟩. Refer to the semantics under the **BUFOVFL** option of the ⟨*receive statement*⟩.

CARRIAGE
Interrogate, T

CARRIAGE is a carriage control variable, and is used to indicate if a carriage return is desired at the completion of the text transmission.

CARRIAGE is **TRUE** if message word [0] . [25:1] is zero.

This bit can be set by the I/O Intrinsics for a data communications file, or by the MCS.

CARRIER
Interrogate, CTR

This ⟨*bit variable*⟩ indicates the state of the Carrier Detect (**CF**) level. It is **TRUE** if and only if the level is up between the modem and line adapter. This variable is always **FALSE** for a direct connect interface.

CARRIER may be used in conjunction with the **TERMINATE DISCONNECT** statement to recover from temporary loss of carrier on switched lines due to extraneous noise on the line.

CHARACTER (CHAR)
Interrogate/Alter, CTR/CTR, 8

CHARACTER is a line-oriented ⟨*byte variable*⟩.

CHARACTER contains the last character **TRANSMIT**ted or **RECEIVE**d on the line, unless otherwise altered by a ⟨*fetch statement*⟩ or an ⟨*assignment statement*⟩.

CONTROL
Interrogate, CTR, 8

This read-only station variable allows access to the **CONTROL** character for the station. This variable contains the value supplied in the most recent **SET CHARACTERS DCWRITE** (type=39) and is initialized to the value specified in the ⟨*station control character statement*⟩.

CONTROLFLAG
Interrogate/Alter, TR/TR

This ⟨*bit variable*⟩ refers to bit 12 in the Error Flag Field of a message, and normally indicates that the station defined control character was received. Refer to the **CONTROL** option of the ⟨*receive statement*⟩.

Note that if this bit is on in a message to be returned to an MCS, and the first character of the message is the control character of the station, the message is returned as a STATION EVENT (CLASS=1). Refer to the B 5000/B 6000/B 7000 Series DCALGOL Language Reference Manual for more information regarding this message.

CRC
Interrogate/Alter, CTR/CTR

In ⟨*request definition*⟩*s* and ⟨*control definition* ⟩*s* that use the Cyclic Redundancy Check system software tests the status of the ⟨*bit variable*⟩ **CRC** before the execution of any ⟨*receive statement* ⟩ or ⟨*transmit statement*⟩. If **CRC** is **TRUE**, the byte (or bytes) transmitted or received are calculated into the Cyclic Redundancy Check stored in the ⟨*byte variable*⟩*s* **CRC[0]** and **CRC[1]**. If **CRC** is **FALSE**, bytes transmitted or received do not affect the Cyclic Redundancy Check.

CRC[{*0 or 1*}**]**
Interrogate/Alter, CTR/CTR, 8

System software uses the ⟨*byte variable*⟩*s* **CRC[0]** and **CRC[1]** as a concatenated 16-bit information field to contain Cyclic Redundancy Check information for those stations whose ⟨*terminal definition*⟩*s* define horizontal parity as **CRC(16).** If the ⟨*bit variable*⟩ **CRC** is **TRUE**, Cyclic Redundancy Check calculation is done using **CRC[0]** and **CRC[1]** as a 16-bit field, and the characters **TRANSMIT**ted or **RECEIVE**d. If **CRC** is **FALSE**, Cyclic Redundancy Check calculation is inhibited.

If a station ⟨*terminal definition*⟩ does not define horizontal parity, then **CRC[0]** and **CRC[1]** can be used as a temporary storage area. It should be pointed out, however, that the values in **CRC[0]** and **CRC[1]** are destroyed by most constructs of the ⟨*terminate statement*⟩. Additionally, since the intended purpose of these variables is storage of parity information, **CRC[0]** and **BCC** address the same byte. **CRC[0]** cannot be used for temporary data storage if the ⟨*control definition*⟩ or ⟨*request definition*⟩ uses **BCC** for Block Check Character accumulation.

CRCERR
Interrogate/Alter, TR/TR

CRCERR references bit 7 in the Error Flag Field of a result message, and conventionally indicates that an error in the Cyclic Redundancy Check occurred while executing a ⟨*receive statement*⟩. Refer to the semantics of the **CRC** option of the ⟨*receive statement*⟩.

DISCONNECT
Interrogate/Alter, TR/TR

DISCONNECT references bit 12 in the Error Flag Field of a message, and indicates that a disconnect occurred on the line while executing a ⟨*request definition*⟩.

END
Interrogate, CTR, 8

This read-only station variable allows access to the end-of-message character for the station. This variable contains the value supplied in the most recent **SET CHARACTERS DCWRITE** (type=39) and is initialized to the value specified in the ⟨*terminal end character statement*⟩.

ENDOFBUFFER
Interrogate/Alter, TR/TR

ENDOFBUFFER references bit 17 in the Error Flag Field of a result message, and is conventionally used by a ⟨*request definition*⟩ to indicate when an overflow of the text buffer has occurred. Refer to the semantics of the **ENDOFBUFFER** option of the ⟨*receive statement*⟩.

FORMATERR
Interrogate/Alter, TR/TR

This bit references bit 10 in the Error Flag Field of a result message, and is conventionally used to indicate that a format error occurred while executing a ⟨*receive statement*⟩. Refer to the **RECEIVE** ⟨*string*⟩ construct of the ⟨*receive statement*⟩.

INHIBITSYNC
Interrogate/Alter, CTR/CTR

INHIBITSYNC is a line-oriented variable that causes actions as described under the ⟨*terminal inhibitsync statement*⟩.

IR
Interrogate, CTR, 10-bit

IR addresses the 10-bit Input Register of the adapter cluster. This register contains hardware related control and data information for a line adapter.

IR can be interrogated using a ⟨*bit number*⟩ specification. ⟨*bit number*⟩*s* for **IR** range from zero through 9. For example, **IR[0]** addresses bit number zero of the Input Register.

Refer to the Burroughs Data Communications Processor FETM number 1041639 for the meaning of the bits in **IR**.

LINE(BUSY)
Interrogate/Alter, CTR/CTR

LINE(BUSY) is a line-oriented control information bit, and is used to allow or inhibit the interruption of the execution of a ⟨*control definition*⟩ or ⟨*request definition*⟩ on a single line. In the case of a full duplex line pair, **LINE(BUSY)** refers to the primary line. If this bit is **TRUE**, it indicates to the DCP operating system that the line is engaged in functions that must not be interrupted. If **FALSE**, it indicates to the DCP operating system that the line can be interrupted to initiate another function. **LINE(BUSY)** can be altered only by the primary line of a full duplex line pair.

A ⟨*control definition*⟩ or ⟨*request definition*⟩ is interrupted when **LINE(BUSY)** is **FALSE** if the DCP receives in its Request Queue a station-oriented DCWRITE message, and **STATION(QUEUED)** is **FALSE** for that station. If the message is a READ - ONCE ONLY (TYPE=34), **STATION** is set to that station index, and control is transferred to the Receive Request for that station. If the message is a WRITE (TYPE=33) DCWRITE message, **STATION** is set to that station index, and control is transferred to the Transmit Request for that station. If the message TYPE is neither of the above, the function associated with the message is executed and control resumes at the beginning of the line ⟨*control definition*⟩, with the value of **STATION** equal to the index of the station for which the function was initiated. (Note that an interruption causes control to leave a ⟨*control definition*⟩ or ⟨*request definition*⟩, and that control is not returned to the point where the interruption occurred.)

LINE(BUSY) is set **TRUE** by system software when:

a. The ⟨*control definition*⟩ is entered,

b. A ⟨*request definition*⟩ is entered, or

c. The line is the primary of a full duplex line pair, **LINE(BUSY)** is **FALSE**, and the auxiliary line executes a ⟨*fork statement*⟩.

Note that if **LINE(BUSY)** is **TRUE** when the auxiliary line of a full duplex line pair executes a ⟨*fork statement*⟩, the ⟨*fork statement*⟩ acts as a no-op.

LINE(QUEUED)
Interrogate/Alter, CTR/CTR

LINE(QUEUED) is a line-oriented variable used to indicate whether or not (**TRUE** or **FALSE**, respectively) a message has been queued for any station on the line. It is set **TRUE** by system software when a message is inserted into an empty Station Queue of a station assigned to the line. It is the programmer's responsibility to set it **FALSE** when appropriate.

LINE(TALLY [tally number])
Interrogate/Alter, CTR/CTR, 8

LINE(TALLY [tally number]) are line-oriented variables for data storage, etc., available to the programmer.

The contents of **LINE(TALLY [tally number])** are cleared to zero when a switched line becomes connected.

An MCS may dynamically alter the line tallies by performing a SET/RESET LINE TOG/TALLY (TYPE=103) DCWRITE request.

LINE(TOG[toggle number])
Interrogate

LINE(TOG[toggle number]) are line-oriented variables for general information storage, etc., available to the programmer.

The contents of **LINE(TOG[toggle number])** are reset to **FALSE** when a switched line becomes connected.

An MCS may dynamically alter the line toggles by performing a SET/RESET LINE TOG/TALLY (TYPE=103) DCWRITE request.

LINEDELETE (LD)
Interrogate, CTR, 8

This read-only station variable allows access to the **LINEDELETE** character for the station. This variable contains the value supplied in the most recent **SET CHARACTERS DCWRITE** (type=39) and is initialized to the value specified in the ⟨*terminal linedelete character statement*⟩.

LINEFEED
Interrogate, T

LINEFEED is a carriage control variable and is **TRUE** when message word [0] . [24:1] is zero. If **TRUE**, **LINEFEED** indicates that a new line is required at the completion of the text transmission. This bit can be set by the I/O Intrinsics for a data communications file, or by the MCS.

LOSSOFCARRIER
Interrogate/Alter, TR/TR

LOSSOFCARRIER references bit 18 in the Error Flag Field of a result message, and is conventionally used to indicate that a loss of carrier occurred while executing a ⟨*receive statement*⟩. Refer to the **LOSSOFCARRIER** option of the ⟨*receive statement*⟩.

MAXSTATIONS
Interrogate, CTR, 8

MAXSTATIONS is a line-oriented ⟨*byte variable*⟩ whose value is the maximum number of stations that can be assigned to the line.

MAXSTATIONS is initialized to the value defined in the ⟨*line maxstations statement*⟩ of the ⟨*line definition*⟩. If the ⟨*line maxstations statement*⟩ does not appear in a ⟨*line definition*⟩, then **MAXSTATIONS** is initialized to the number of stations listed in the ⟨*line station statement*⟩. If neither statement appears, **MAXSTATIONS** is zero.

Within a ⟨*control definition*⟩, the valid range of values which may be assigned to the ⟨*byte variable*⟩ **STATION** is between zero and **MAXSTATIONS** -1, inclusive.

NAKFLAG
Interrogate/Alter, TR/TR

NAKFLAG references bit 11 in the Error Flag Field of a result message, and conventionally indicates that a transmission was NAKed by the terminal. This bit is not set by system software, and its use is at the option of the programmer.

NAKONSELECT
Interrogate/Alter, TR/TR

NAKONSELECT references bit 16 of the Error Flag Field of a result message, and is conventionally used to indicate that a Transmit Request was NAKed when it attempted to select the terminal. This bit is not set by system software, and its use is at the option of the programmer.

NOSPACE
Interrogate, CTR

NOSPACE is a global variable that, when **TRUE**, indicates that a "no space" condition exists in the available space pool. **NOSPACE** is set by system software when the condition exists, and reset when the condition no longer exists.

PAGE
Interrogate, T

PAGE is a carriage control variable, and conventionally indicates whether a new page is required for the output device. For example, on a screen device, **PAGE = TRUE** could indicate to the Transmit Request that a home/clear sequence should be transmitted before or after the text is transmitted to the terminal. Refer to **PAPERMOTION**.

PAGE is set **TRUE** is message word [0].[26:1] = 1.

PAPERMOTION
Interrogate, T

PAPERMOTION is a carriage control variable that is conventionally used to indicate whether carriage control is desired before or after the message text is transmitted. If message word [0].[30:1] = 1, **PAPERMOTION** is set **TRUE**, and carriage control should be done before the text is transmitted; otherwise, carriage control after the text is transmitted.

PARITY
Interrogate/Alter, TR/TR

PARITY references bit 6 of the Error Flag Field in a result message, and indicates that a vertical parity error was detected when executing a ⟨*receive statement*⟩. Refer to the **PARITY** option of the ⟨*receive statement*⟩.

RETRY
Interrogate/Alter, CTR/CTR, 8

RETRY is a station-oriented variable, and is referred to as DCP RETRY.

The purpose of DCP RETRY is to record the number of attempts a ⟨*request definition*⟩ has made to communicate with a terminal but failed as the result of some abnormal condition. Conventionally, the NDL programmer decrements **RETRY** (i.e., DCP RETRY) by one for each unsuccessful attempt at an operation until **RETRY** equals zero, then executes a **TERMINATE ERROR**.

When a ⟨*request definition*⟩ is initiated by the DCP, DCP RETRY is implicitly set to an initial value called DCP INITIAL RETRY. The default value of DCP INITIAL RETRY is specified by the NDL program in the ⟨*station retry statement*⟩.

By using the Message Retry Field in the message header (message word [2].[47:8]), the MCS can control the value assigned to DCP INITIAL RETRY, and therefore, is the initial value of DCP RETRY. If the Message Retry Field is 255, the value specified in the ⟨*station retry statement*⟩ assigned to DCP INITIAL RETRY, otherwise the value of the Message Retry Field is assigned to DCP INITIAL RETRY. The NDL program can restore the value of DCP RETRY to the value of DCP INITIAL RETRY at any time by executing the **INITIALIZE RETRY** construct.

All forms of the ⟨*terminate statement*⟩ which result in a message being returned to an MCS cause the current value of DCP RETRY to be stored in the Message Retry Field of the result message.

SEQERR
Interrogate/Alter, TR/TR

SEQERR references bit 14 in the Error Flag Field of a result message, and conventionally indicates that a sequence number overflow occurred as the result of the execution of an **INCREMENT SEQUENCE** construct. Refer to the *⟨increment statement⟩*.

SEQUENCE
Interrogate/Alter, CTR/CTR

SEQUENCE is a station-oriented bit that indicates whether or not (**TRUE** or **FALSE**, respectively) a *⟨request definition⟩* is to perform automatic sequencing. **SEQUENCE** is controlled by a SET/RESET SEQUENCE MODE (TYPE=49) DCWRITE from the MCS. **SEQUENCE** can be set **FALSE** by the NDL program but can be set **TRUE** only by the controlling MCS. **SEQUENCE** can be set **TRUE** only if the *⟨request definition⟩* for a terminal employs sequence number constructs such as **TRANSMIT SEQUENCE**, **INCREMENT SEQUENCE**, and **STORE SEQUENCE**. Use of automatic sequencing is the option and responsibility of the NDL programmer.

SKIP
Interrogate/T

SKIP is a carriage control variable. **SKIP** is used in conjunction with **SKIPCONTROL** to indicate a "skip to channel N" on an output device. If message word [0].[27:1] = 1, **SKIP** is set **TRUE** and **SKIPCONTROL** contains the channel number to skip to. Both **SKIP** and **SKIPCONTROL** can be set by the I/O Intrinsics for a data communications file, or by the MCS.

SKIPCONTROL
Interrogate, T, 8

SKIPCONTROL is used in conjunction with the *⟨bit variable⟩s* **SKIP** and **SPACE**. If **SKIP** is **TRUE**, then **SKIPCONTROL** applies to **SKIP**. If **SPACE** is **TRUE**, **SKIPCONTROL** applies to **SPACE**. If neither are **TRUE**, **SKIPCONTROL** is undefined. Both **SKIP** and **SPACE** should not be **TRUE** concurrently. For a description of the function of this byte, refer to **SPACE** and **SKIP**. **SKIPCONTROL** is transferred to the Transmit Request in the message header of a WRITE (TYPE=33) DCWRITE in message word [0].[39:8], and can be set by the I/O Intrinsics for a data communications file, or by the MCS.

SPACE
Interrogate, T

SPACE is a carriage control variable. **SPACE** is used in conjunction with **SKIPCONTROL** to indicate the number of vertical lines to skip. If message word [0].[28:1] = 1, **SPACE** is set **TRUE** and **SKIPCONTROL** indicates the number of lines to skip. **SPACE** and **SKIPCONTROL** can be set by the I/O Intrinsics for a data communications file, or by the MCS.

STATION
Interrogate/Alter, CTR/C, 8

STATION is a line-oriented ⟨*byte variable*⟩ used in a ⟨*control definition*⟩ of a multi-station line to select a particular station with which the ⟨*control definition*⟩ wishes to interact. That is, to access the variables of a particular station, or **INITIATE** the Receive Request or Transmit Request of a station, the station index value associated with the station must be stored in **STATION**.

A station index value is associated with each station that is assigned to a logical line. At DCP initialization time, station index values are assigned sequentially, beginning at zero, to each station on a given line in the order that the stations were named in the ⟨*line station statement*⟩ of the ⟨*line definition*⟩.

After DCP initialization, an MCS can cause a station to be logically added to a line. When this occurs, a station index value becomes associated with the station. An MCS can also cause the logical removal of a station from a line. After such action, the station index value that was associated with the station no longer references a valid station. Thus, after DCP initialization, "holes" can exist in the sequence of valid station index values for a given line. A station index value can be "tested" to determine if it references a valid station by interrogating the **STATION(VALID)** ⟨*bit variable*⟩.

There is a maximum valid station index value associated with each line. That value is determined either by the ⟨*line maxstations statement*⟩ or by the ⟨*line station statement*⟩. (Refer to the ⟨*line maxstations statement*⟩ for more information.) This value can be obtained in a ⟨*control definition*⟩ by interrogating **MAXSTATIONS**.

STATION(ENABLED)
Interrogate, CTR

This is a station-oriented ⟨*bit variable*⟩ which refers to the "enabled" state of a station. When this variable is **TRUE**, the station is enabled for input, and the station Receive Request can be invoked. If **STATION(ENABLED)** is **FALSE**, the station is disabled for input, and attempts to invoke the Receive Request will be disallowed.

The setting of **STATION(ENABLED)** is initially defined by the ⟨*station enableinput statement*⟩ in the station definition, and may be altered by an MCS via the ENABLE INPUT (TYPE=35) and DISABLE INPUT (TYPE=36) DCWRITE.

STATION(FREQUENCY)
Interrogate, CTR, 8

STATION(FREQUENCY) is a station-oriented ⟨*byte variable*⟩, and is conventionally used to contain a relative polling frequency for polled stations. The initial value for **STATION(FREQUENCY)** is supplied by the ⟨*station frequency statement*⟩ for a station. It can be altered by an MCS via the ENABLE INPUT (TYPE=35) DCWRITE. Refer to the ⟨*station frequency statement*⟩.

STATION(QUEUED)
Interrogate, CTR

STATION(QUEUED) is a station-oriented variable that indicates whether or not (**TRUE** or **FALSE**, respectively) there are any messages (output or enableinput) in the station queue. Note that if this variable is **FALSE**, the execution of an **INITIATE REQUEST** construct acts as a no-op.

STATION(READY)
Interrogate, CTR

If **STATION(READY)** is **TRUE**, the station associated with the station index stored in **STATION** is logically ready. No function (e.g., a Transmit Request or Receive Request) can be **INITIATE**d for the station if it is not ready. Stations can become not-ready as the result of the execution of a **TERMINATE ERROR** in one of its ⟨*request definition*⟩*s* or as the result of the MCS executing a MAKE STATION NOT-READY (TYPE=37) DCWRITE.

STATION(TALLY)
Interrogate/Alter, CTR/CTR, 8

STATION(TALLY) is a station-oriented ⟨*byte variable*⟩ and is a general purpose variable which may be used by the NDL program for data storage. The initial value of **STATION(TALLY)** is zero. Note that **STATION(TALLY)** differs from **TALLY** [⟨*tally number*⟩] in that it cannot be directly **STORE**d in a message header.

STATION(VALID)
Interrogate, CTR

The **STATION(VALID)** bit indicates whether or not (**TRUE** or **FALSE**, respectively) there is a valid station associated with the station index value stored in **STATION**. Refer to the ⟨*byte variable*⟩ **STATION**.

STOPBIT
Interrogate/Alter, TR/TR

STOPBIT references bit 1 in the Error Flag Field of a result message, and conventionally indicates that a stop bit error was detected while executing a ⟨*receive statement*⟩. Refer to the **STOPBIT** option of the ⟨*receive statement*⟩.

SYNCS
Interrogate/Alter, CTR/CTR

SYNCS is a synonym for **INHIBITSYNC**. Refer to the **INHIBITSYNC** description.

SYSTEMDUMPING
Interrogate, CTR

SYSTEMDUMPING is a global variable that is **TRUE** while the main system processors are taking a non-fatal memory dump. Upon completion of the dump, the variable is automatically reset.

This variable may be used in conjunction with **NOSPACE** to indicate the reason for the **NOSPACE** condition.

TALLY [⟨*tally number*⟩]
Interrogate/Alter, CTR/CTR, 8

TALLY [0] through **TALLY [255]** are general purpose station-oriented ⟨*byte variable*⟩s. They can be used for storage of 8-bit quantities such as counters, characters, etc. When the DCP is initialized, the **TALLY**s are initially zero unless a value is specified in a ⟨*station initialize statement*⟩. **TALLY**s may be initialized directly from a message header (message word [3].[23:24]) by utilizing the **INITIALIZE TALLY** [⟨*tally number*⟩] construct, thereby enabling an MCS to supply additional information to the DCP. The DCP can likewise transfer the value of a **TALLY** back to an MCS in a result message by utilizing the ⟨*store statement*⟩. Once a **TALLY** has been assigned a value, that **TALLY** retains that value until explicitly altered by the NDL program.

NOTE

Only **TALLY [0]** – **TALLY [2]**, and **TOG [0]** – **TOG [7]**, can be **INITIALIZED** from an **MCS** message **HEADER**, or **STORE**d there.

NOTE

To conserve space in station and line tables the **NDL PROGRAMMER** should use low **TOG** and **TALLY** numbers. The largest **TOG/TALLY** number defines the amount of space allocated in the Line/Station Table.

TIMEOUT
Interrogate/Alter, TR/TR

TIMEOUT references bit 0 (zero) of the Error Flag Field in a result message, and conventionally indicates that a timeout occurred while executing a ⟨*receive statement*⟩. Refer to the **TIMEOUT** option of the ⟨*receive statement*⟩.

TOG [⟨*toggle number*⟩]
Interrogate/Alter, CTR/CTR

TOG [0] through **TOG [255]** are general purpose station-oriented ⟨*bit variable*⟩s, often referred to as toggles. They can be used for storage of logical values (**TRUE** and **FALSE**). When the DCP is initialized, the value of the toggles is set to the value specified in the ⟨*station initialize statement*⟩ or, if such initialization is not specified, the initial value will be **FALSE**. Toggles can be assigned a value directly from a

message header (message word [1].[39:8]) by utilizing the **INITIALIZE TOG** [⟨*toggle number*⟩] construct. Toggles can be stored into a result message by utilizing the ⟨*store statement*⟩. Once a toggle has been assigned a value, that toggle retains that value until explicitly altered by the NDL program.

NOTE

Only **TALLY [0] - TALLY [2]**, and **TOG [0] - TOG [7]** can be **INITIALIZE**d from an MCS message **HEADER**, or **STORE**d there.

NOTE

To conserve space in station and line tables the **NDL PROGRAMMER** should use low **TOG** and **TALLY** numbers. The largest **TOG/TALLY** number defines the amount of space allocated in the Line/Station Table.

TOGGLES [⟨*toggles number*⟩]
Interrogate/Alter, CTR/CTR, 8

The subscripted byte variable, "**TOGGLES**" allows the **NDL** programmer to address eight **TOG**s as one byte variable. The subscript must be between 0 and 31 inclusive. "**TOGGLES[X]**" references "**TOG[X*8]**" thru "**TOG[X*8+7]**".

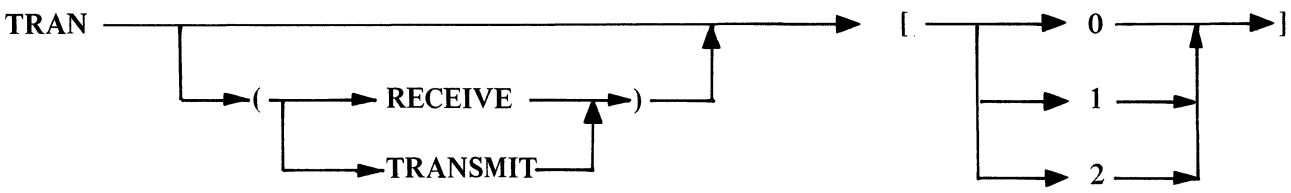
Example:

```
TOGGLES[0] = 0.    % sets TOGs 0-7 to false
TOGGLES[1] = 255.  % set TOGs 8-15 to true
```

TOGS
Interrogate/Alter, CTR/CTR, 8

TOGS addresses the eight ⟨*bit variable*⟩*s* **TOG[0]** through **TOG [7]**. For example, **TOGS = 4"FF"** sets **TOG[0]** through **TOG [7] TRUE. TOG [0]** is considered the low-order bit, and **TOG [7]** the high-order bit.

TRAN
Interrogate/Alter, CTR/CTR, 8



Example:

TRAN (RECEIVE) [1] = TALLY [0]

These bytes allow access to a stations transmission numbers. They are grouped as two sets of three bytes each, one set for transmit and one for receive. Within each group, the three bytes are numbered from zero. Byte zero represents the least significant byte. The "**RECEIVE**" or "**TRANSMIT**" part may be omitted, in which case the receive bytes will be selected in a receive request or control definition and the transmit bytes in a transmit request.

Within each station table, the transmission number bytes are stored from left to right as follows:

TRAN(TRANSMIT) [2]
TRAN(TRANSMIT) [1]
TRAN(TRANSMIT) [0]
TRAN(RECEIVE) [2]
TRAN(RECEIVE) [1]
TRAN(RECEIVE) [0]

Unless they are explicitly modified by the **NDL** program, all six **TRAN** bytes are **EBCDIC** numeric characters, even if the terminal requires fewer than three transmission digits. Note that the transmission number must be **EBCDIC** numeric for the **INCREMENT TRAN** construct to function properly.

TRANERR
Interrogate/Alter, TR/TR

TRANERR references bit 9 in the Error Flag Field of a result message, and is conventionally used to indicate that a transmission number error occurred. Refer to the **TRAN** option of the ⟨*receive statement*⟩.

TRANSMITMODE
Interrogate, CTR

TRANSMITMODE reflects the state of the Request to Send (**CA**) level of an adapter connected to a modem. When **TRUE**, it indicates the adapter is in transmit mode. When **FALSE**, it indicates the adapter is in receive mode.

WRU
Interrogate, CTR, 8

This read-only station variable allows access to the **WRU** character for the station. This variable contains the value supplied in the most recent **SET CHARACTERS DCWRITE** (type=39) and is initialized to the value specified in the ⟨*terminal WRU character statement*⟩.

WRUFLAG
Interrogate/Alter, TR/TR

WRUFLAG references bit 13 in the Error Flag Field of a result message. If this bit is **TRUE** upon termination of a ⟨*request definition*⟩, the result message is returned to the MCS as a STATION EVENT (CLASS = 1) message.

APPENDIX A

RESERVED WORDS

The following is a complete list of reserved words used in the Network Definition Language. These words have special meaning to the compiler and cannot be used as ⟨*identifier*⟩*s* or in any manner other than their defined meaning. Any synonym of a reserved word is shown adjacent to the word, in parentheses.

ABORT	
ADAPTER	(ADAPTOR)
ADAPTOR	(ADAPTER)
ADDERR	
ADDRESS	
AI	
ALTERNATE	
ANSWER	
APPLICATION	
ASCII	
ASC63	
ASC67	
ASC68	
ASL	
ASR	
AUX	(AUXILIARY)
AUXILIARY	(AUX)
BACKSPACE	(BKSP)
BAUDOT	
BCC	
BCCERR	(CRCERR)
BCD	
BCL	
BEGIN	
BINARY	
BKSP	(BACKSPACE)
BLKN	
BLKNERR	
BLOCK	(BLOCKED)
BLOCKED	(BLOCK)
BREAK	
BUFFER	
BUFOVFL	
BUSY	
CARRIAGE	
CARRIER	
CHAR	(CHARACTER)
CHARACTER	(CHAR)
CLEAR	
CLUSTERS	
CODE	
CONNECTION	
CONSTANT	
CONTINUE	
CONTROL	
CRC	
CRCERR	(BCCERR)
DCP	
DEFAULT	
DEFINE	
DELAY	
DIALIN	
DIALOUT	

RESERVED WORDS (Cont)

DIFFERENT	
DIRECT	
DISCONNECT	
DOWN	
DUPLEX	
DYNAMIC	
EBCDIC	
ELSE	
ENABLED	
ENABLEINPUT	
END	
ENDOFBUFFER	
ENDOFNUMBER	
EQ	(EQL)
EQL	(EQ)
ERROR	
ERRORFLAGS	
EVEN	
EXCHANGE	
FALSE	
FAMILY	
FETCH	
FILE	
FINISH	
FOR	
FORK	
FORMAT	
FORMATERR	
FREQUENCY	
GE	(GEQ)
GEQ	(GE)
GETSPACE	
GO	
GT	(GTR)
GTR	(GT)
HOME	
HORIZONTAL	
ICTDELAY	
IDLE	
IF	
ILLEGALCHR	
INCREMENT	
INHIBITSYNC	(SYNCS)
INITIALIZE	
INITIATE	
INPUT	
IR	
LD	(LINEDELETE)
LE	(LEQ)
LEQ	(LE)
LINE	
LINEDELETE	(LD)
LINEFEED	
LOCALTABLES	
LOGICALACK	
LOGIN	
LOSSOFCARRIER	
LS	(LSS)
LSS	(LS)
MAXINPUT	
MAXOUTPUT	
MAXSTATIONS	
MCS	
MEMORY	
MICRO	

RESERVED WORDS (Cont)

MILLI		REQUEST	
MIN		RETRY	
MODE		RETURN	
MODEM		ROL	
MSGSPACE		ROR	
MYUSE		SCREEN	
NAKFLAG		SEC	
NAKONSELECT		SECUREDLINES	
NE	(NEQ)	SCURITY	
NEQ	(NE)	SEQERR	
NOINPUT		SEQUENCE	
NOISEDELAY		SHIFT	
NORMAL		SKIP	
NOSPACE		SKIPCONTROL	
NOT		SPACE	
NULL		SPO	
ODD		STANDARD	
ON		STATION	
OPTION(S)		STOPBIT	
OUTPUT		STORE	
PAGE		SUM	
PAPERMOTION		SYNCS	(INHIBITSYNC)
PARITY		SYSTEMDUMPING	
PASSIVE		TAB	
PAUSE		TALLY	
PHONE		TASK	
PTTC6I		TERMINAL	
QUEUED		TERMINATE	
READY		TEXT	
RECEIVE		THEN	
RECEIVEDELAY		TIMELIMIT	
REMOTE		TIMEOUT	

RESERVED WORDS (Cont)

TO
TOG
TOGGLES
TOGS
TRAN
TRANERR
TRANSLATETABLE
TRANSLATOR
TRANSMISSION
TRANSMIT
TRANSMITDELAY
TRANSMITMODE
TRUE
TURNAROUND
TYPE
UP
USER
VALID
WAIT
WIDTH
WRAPAROUND
WRU
WRUFLAG
XOR

APPENDIX B

TRANSMISSION CODES

BAUDOT CODE

Bits b6					0	0	1	1
Bits b5					0	1	0	1
b4	b3	b2	b1	Column / Row	0	1	2	3
0	0	0	0	0	BLK	T	BLK	5
0	0	0	1	1	E	Z	3	"
0	0	1	0	2	LF	L	LF	3/4)
0	0	1	1	3	A	W	-	2
0	1	0	0	4	SPACE	H	SPACE	DIAMOND
0	1	0	1	5	S	Y	BELL	6
0	1	1	0	6	I	P	8	0
0	1	1	1	7	U	Q	7	1
1	0	0	0	8	CR	O	CR	9
1	0	0	1	9	D	B	$	5/8 ?
1	0	1	0	10 (A)	R	G	4	&
1	0	1	1	11 (B)	J	FIGS	'	FIGS
1	1	0	0	12 (C)	N	M	7/8 ,	.
1	1	0	1	13 (D)	F	X	1/4 !	/
1	1	1	0	14 (E)	C	V	1/8 :	3/8 ;
1	1	1	1	15 (F)	K	LTRS	1/2 (	LTRS

DATA REPRESENTATION

EBCDIC GRAPHIC	BCL GRAPHIC	HEX.	EBCDIC INTERNAL	DECIMAL VALUE	EBCDIC CARD CODE	OCTAL	BCL INTERNAL	BCL EXTERNAL	BCL CARD CODE	USASCII X3.4-1967
Blank		40	0100 0000	64	No Punches	60	11 0000	01 0000	No Punches	010 0000
[		4A	0100 1010	74	12 8 2	33	01 1011	11 1100	12 8 4	101 1011
.		4B	0100 1011	75	12 8 3	32	01 1010	11 1011	12 8 3	010 1110
<		4C	0100 1100	76	12 8 4	36	01 1110	11 1110	12 8 6	011 1100
(		4D	0100 1101	77	12 8 5	35	01 1101	11 1101	12 8 5	010 1000
+		4E	0100 1110	78	12 8 6			11 1010		010 1011
|	←	4F	0100 1111	79	12 8 7	37	01 1111	11 1111	12 8 7	111 1100
&		50	0101 0000	80	12	34	01 1100	11 0000	12	010 0110
]		5A	0101 1010	90	11 8 2	76	11 1110	01 1110	0 8 6	101 1101
$		5B	0101 1011	91	11 8 3	52	10 1010	10 1011	11 8 3	010 0100
*		5C	0101 1100	92	11 8 4	53	10 1011	10 1100	11 8 4	010 1010
)		5D	0101 1101	93	11 8 5	55	10 1101	10 1101	11 8 5	010 1001
;		5E	0101 1110	94	11 8 6	56	10 1110	10 1110	11 8 6	011 1011
¬	≤	5F	0101 1111	95	11 8 7	57	10 1111	10 1111	11 8 7	
-		60	0110 0000	96	11	54	10 1100	10 0000	11	101 1111
/		61	0110 0001	97	0 1	61	11 0001	01 0001	0 1	010 1111
,		6B	0110 1011	107	0 8 3	72	11 1010	01 1011	0 8 3	010 1100
%		6C	0110 1100	108	0 8 4	73	11 1011	01 1100	0 8 4	010 0101
¦	≠	6D	0110 1101	109	0 8 5	74	11 1100	01 1010	0 8 2	010 1101
>		6E	0110 1110	110	0 8 6	16	00 1110	00 1110	8 6	011 1110
?		6F	0110 1111	111	0 8 7	14	00 1100	00 0000	*	011 1111
:		7A	0111 1010	122	8 2	15	00 1101	00 1101	8 5	011 1010
#		7B	0111 1011	123	8 3	12	00 1010	00 1011	8 3	010 0011
@		7C	0111 1100	124	8 4	13	00 1011	00 1100	8 4	100 0000
'	≥	7D	0111 1101	125	8 5	17	00 1111	00 1111	8 7	010 0111
=		7E	0111 1110	126	8 6	75	11 1101	01 1101	0 8 5	011 1101
"		7F	0111 1111	127	8 7	77	11 1111	01 1111	0 8 7	010 0010

EBCDIC GRAPHIC	BCL GRAPHIC	HEX.	EBCDIC INTERNAL	DECIMAL VALUE	EBCDIC CARD CODE			OCTAL	BCL INTERNAL	BCL EXTERNAL	BCL CARD CODE		USASCII X3.4-1967
(+)PZ	+	C0	1100 0000	192	12	0		20	01 0000	11 1010	12	0	
A		C1	1100 0001	193	12	1		21	01 0001	11 0001	12	1	100 0001
B		C2	1100 0010	194	12	2		22	01 0010	11 0010	12	2	100 0010
C		C3	1100 0011	195	12	3		23	01 0011	11 0011	12	3	100 0011
D		C4	1100 0100	196	12	4		24	01 0100	11 0100	12	4	100 0100
E		C5	1100 0101	197	12	5		25	01 0101	11 0101	12	5	100 0101
F		C6	1100 0110	198	12	6		26	01 0110	11 0110	12	6	100 0110
G		C7	1100 0111	199	12	7		27	01 0111	11 0111	12	7	100 0111
H		C8	1100 1000	200	12	8		30	01 1000	11 1000	12	8	100 1000
I		C9	1100 1001	201	12	9		31	01 1001	11 1001	12	9	100 1001
(!)MZ	x	D0	1101 0000	208	11	0		40	10 0000	10 1010	11	0	010 0001
J		D1	1101 0001	209	11	1		41	10 0001	10 0001	11	1	100 1010
K		D2	1101 0010	210	11	2		42	10 0010	10 0010	11	2	100 1011
L		D3	1101 0011	211	11	3		43	10 0011	10 0011	11	3	100 1100
M		D4	1101 0100	212	11	4		44	10 0100	10 0100	11	4	100 1101
N		D5	1101 0101	213	11	5		45	10 0101	10 0101	11	5	100 1110
O		D6	1101 0110	214	11	6		46	10 0110	10 0110	11	6	100 1111
P		D7	1101 0111	215	11	7		47	10 0111	10 0111	11	7	101 0000
Q		D8	1101 1000	216	11	8		50	10 1000	10 1000	11	8	101 0001
R		D9	1101 1001	217	11	9		51	10 1001	10 1001	11	9	101 0010
¢		E0	1110 0000	224	0	8	2			00 0000			
S		E2	1110 0010	226	0	2		62	11 0010	01 0010	0	2	101 0011
T		E3	1110 0011	227	0	3		63	11 0011	01 0011	0	3	101 0100
U		E4	1110 0100	228	0	4		64	11 0100	01 0100	0	4	101 0101
V		E5	1110 0101	229	0	5		65	11 0101	01 0101	0	5	101 0110
W		E6	1110 0110	230	0	6		66	11 0110	01 0110	0	6	101 0111
X		E7	1110 0111	231	0	7		67	11 0111	01 0111	0	7	101 1000
Y		E8	1110 1000	232	0	8		70	11 1000	01 1000	0	8	101 1001
Z		E9	1110 1001	233	0	9		71	11 1001	01 1001	0	9	101 1010
0		F0	1111 0000	240	0			00	00 0000	00 1010	0		011 0000
1		F1	1111 0001	241	1			01	00 0001	00 0001	1		011 0001
2		F2	1111 0010	242	2			02	00 0010	00 0010	2		011 0010
3		F3	1111 0011	243	3			03	00 0011	00 0100	3		011 0100
4		F4	1111 0100	244	4			04	00 0100	00 0100	4		011 0100

DATA REPRESENTATION (Cont)

EBCDIC GRAPHIC	BCL GRAPHIC	HEX.	EBCDIC INTERNAL	DECIMAL VALUE	EBCDIC CARD CODE	OCTAL	BCL INTERNAL	BCL EXTERNAL	BCL CARD CODE	USASCII X3.4-1967
5		F5	1111 0101	245	5	05	00 0101	00 0101	5	011 0101
6		F6	1111 0110	246	6	06	00 0110	00 0110	6	011 0110
7		F7	1111 0111	247	7	07	00 0111	00 0111	7	011 0111
8		F8	1111 1000	248	8	10	00 1000	00 1000	8	011 1000
9		F9	1111 1001	249	9	11	00 1001	00 1001	9	011 1001

NOTES

1. **EBCDIC** 0100 1110 also translates to **BCL** 11 1010.

2. **EBCDIC** 1100 1111 is translated to **BCL** 00 0000 with an additional flag bit on the most-significant bit line (8th bit). This function is used by the unbuffered printer to stop scanning.

3. **EBCDIC** 1110 0000 is translated to **BCL** 00 0000 with an additional flag bit on the next to most significant bit line (8th bit). As the print drums have 64 graphics and spaces, this signal can be used to print the 64th graphic. The 64th graphic is a "**CR**" for **BCL** drums and a "¢" for **EBCDIC** drums.

4. The remaining 189 **EBCDIC** codes are translated to **BCL** 00 0000 (?code).

5. The **EBCDIC** graphics and **BCL** graphics are the same except as follows:

BCL		EBCDIC	
≥		'	(single quote)
x	(multiply)	!	
≤		¬	(not)
≠		—	(underscore)
←		|	(or)

PTTC/6 CODE

b7 →					0	0	0	0	1	1	1	1
Bit b6 →					0	0	1	1	0	0	1	1
b5 →					0	1	0	1	0	1	0	1
b4	b3	b2	b1	Row \ Column →	0	1	2	3	4	5	6	7
0	0	0	0	0	SPACE	@	-	+ / &	SPACE	DELTA	BACK/	<
0	0	0	1	1	1	/	j	a	>	QUES	J	A
0	0	1	0	2	2	s	k	b	)	S	K	B
0	0	1	1	3	3	t	l	c	;	T	L	C
0	1	0	0	4	4	u	m	d	SBLANK	U	M	D
0	1	0	1	5	5	v	n	e	(	V	N	E
0	1	1	0	6	6	w	o	f	:	W	O	F
0	1	1	1	7	7	x	p	g	"	X	P	G
1	0	0	0	8	8	y	q	k	*	Y	Q	H
1	0	0	1	9	9	z	i	i	[	Z	R	I
1	0	1	0	10(A)	0	#	M7	PZ	]	GRPMRK	GAMMA	SQ. RT
1	0	1	1	11(B)	= / +	,	$	.	SEGMARK	,	V.BAR	.
1	1	0	0	12(C)	PN	BY	RFS	PF	PN	BY	RES	PF
1	1	0	1	13(D)	Rs	LF	NL	HT	RS	LF	NL	HT
1	1	1	0	14(E)	UC	EOR	BS	LC	UC	EOB	BS	LC
1	1	1	1	15(F)	EOT	PRE	IL	DEL	EOT	PRF	IL	DEL

APPENDIX C

SOURCE INPUT FORMAT AND CODING FORM

SOURCE INPUT FORMAT

An NDL source program is represented by a set of ordered external records. The external records could come from cards, tape, disk, remote device, or a combination of these. The source information on any given record must be divided into two areas. Character positions 1 – 72 are assumed to contain elements of the Network Definition Language (described in sections 2 through 6 of this manual) for compilation by the compiler. Character positions 73 through 80 are assumed to contain information regarding the sequence of the input record; specifically, this area is for sequence numbers. Sequence numbers are optional.

There is no fixed format for source information in character positions 1 through 72. This information can appear in a free format form, with the following exceptions: elements contained on the card must comply with any syntactical restrictions, and syntactical items cannot be continued from record to the next. For example, the reserved word **TERMINAL** cannot begin on one source record and continue on the next.

CODING FORM

To facilitate keypunching, as well as to provide the programmer with a suggested format to follow in writing his source program, printed programming forms are often used. An example of such a form appears on the following page.

CODING FORM

PROGRAM ID | PROJECT NO. | COST CENTER | PAGE OF

PROGRAMMER | DATE

SEQUENCE NO.

1 5 10 15 20 25 30 35 40 45 50 55 60 65 70 73 80

SOURCE PROGRAM CODE
(columns 1-72)

SOURCE PROGRAM SEQUENCE NUMBERS
(The sequence numbers in columns 73-80 are not executed, but are reproduced on the source printout.)

1 5 10 15 20 25 30 35 40 45 50 55 60 65 70 73 80

SYMBOLS TO USE 1 FOR DIGIT ONE, I FOR LETTER i, 0 FOR DIGIT ZERO, ϕ FOR LETTER O, X FOR LETTER X, ⊗ FOR MULTIPLY OPERATOR

APPENDIX D

COMPILE-TIME OPTIONS

COMPILER CONTROL STATEMENTS

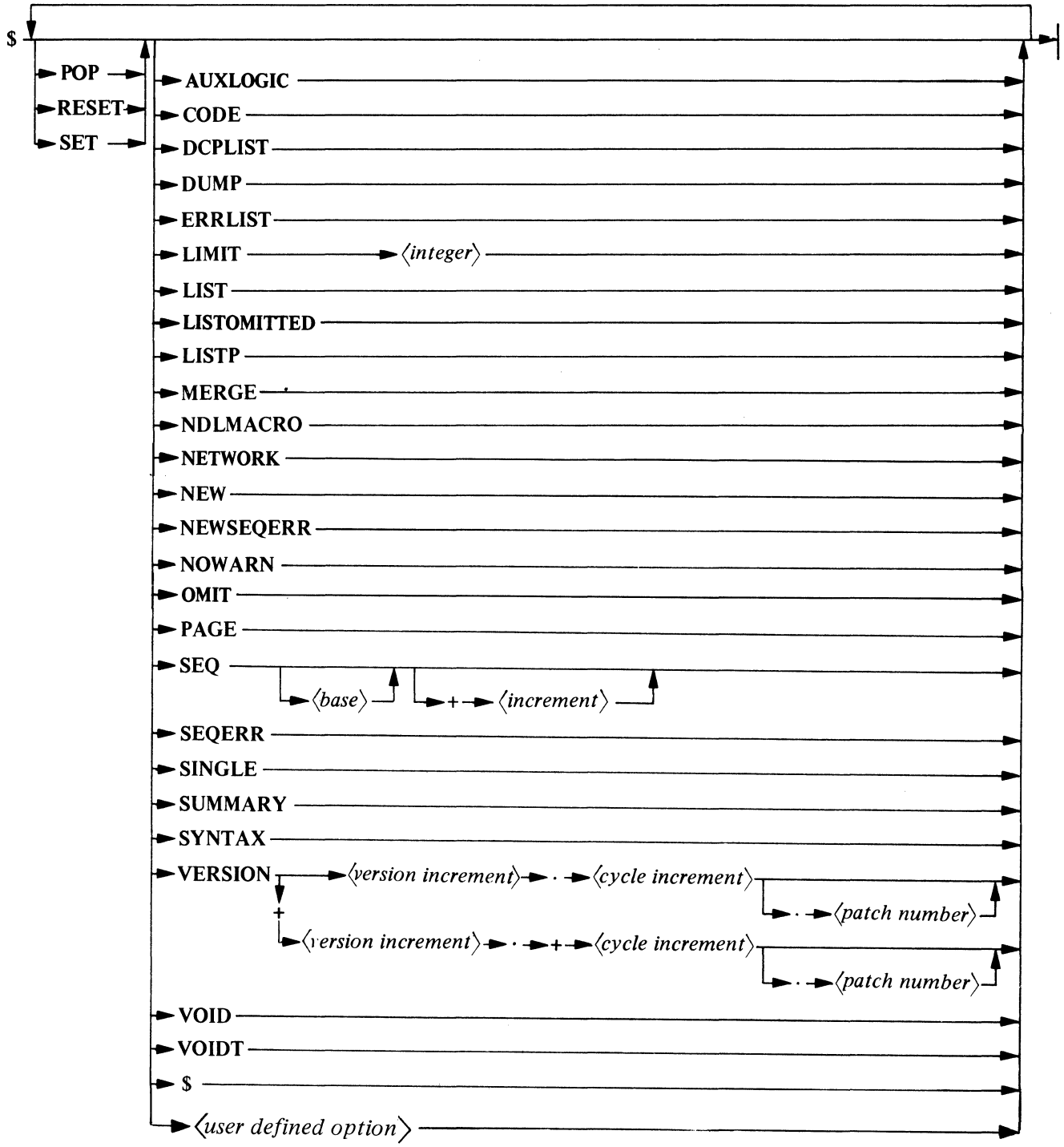
The user is provided with the compile-time ability to control the manner in which the compiler processes the source input that it accepts. The user can specify the manner in which the compiler is to receive the source input, the consequences of certain syntax errors, and the form of the generated compiler output. The compiler control statement is the medium by which these constraints are communicated to the compiler. Such statements are entered into the compiler by cards in the same manner as source language statements. Compiler control statements, entered as input to the compiler via option control cards, can occur at any point in the compiler input files and must contain only compiler control information.

An option control card is identified by the appearance of a dollar sign ($) in the first or second column of the card. If the $ is placed in card column 2, the option control card image is placed in the updated symbolic file (**NEWTAPE**) if such a file is generated. Compilation control information is punched in the succeeding columns through column 72, with an eight-digit sequence number in columns 73 through 80. All blanks in columns 73 through 80 represent the lowest-value sequence number. An option control card with no other compiler information causes the card image in the secondary input file that has the same sequence number to be ignored.

The basic element of compiler control information is the compiler option, which can be invoked by the appearance of its name on an option control card. Two mutually exclusive states are associated with the majority of these options: **SET** and **RESET**; various compiler functions are dependent upon the states of such options. Default states are assigned to these compiler options, and the desired state of such an option can be specified on an option control card. Such option control cards can also contain arguments associated with the option. The balance of compiler options are parameter options with which no states are associated. The functions performed by these latter options are initiated by the appearance on an option control card of the appropriate option name and any related arguments.

OPTION CONTROL CARDS

Syntax



Semantics

The purpose of a compiler control statement is the assignment of a desired value or state (**SET** or **RESET**) to an indicated compiler option(s). Such a control statement must begin with either an explicit or an implicit option action. An explicit option action is defined as one of the following mnemonics: **SET**, **RESET**, or **POP**.

An implicit option action is indicated when a compiler control statement contains only the names of options and no explicit option action. In the latter case, all options named in the compiler control statement are assigned the state **SET**, and all other options are assigned the state **RESET**.

If a compiler control statement begins with the option action **SET**, the options following the option action are assigned the state **SET**; the states of all other options are unchanged. If the compiler control statement begins with the option action **RESET**, the options following the option action are assigned the state **RESET**; the states of all other options are unchanged. If the specified option action is **POP**, then the options have not been changed previously from their default states. The states of all other options are unchanged. The following statements are examples of compiler control statements employing the **SET**, **RESET**, and **POP** option actions.

```
$ SET LIST SINGLE
$ RESET VOID
$ POP NEW NEWSEQERR
$ SET SEQ 0+100
```

An option that has a default state of **RESET** is initially assigned a 48-bit stack word filled with zeros; an option that has a default state of **SET** is initially assigned a 48-bit stack word with a 1 on top and zeros in the remaining positions. The top stack position denotes the state of the option at any time. Each **SET** option action causes the stacks allocated to the designated standard options to be pushed down one bit and a 1 to be placed at the top of each of these stacks. Each **RESET** causes the appropriate option stacks to be pushed down one bit and a 0 to be placed at the tops of these stacks. **POP** causes the stacks corresponding to the designated options to be **POP**ped up one bit, causing the associated options to revert to their immediate previous states. Since the size of these option stacks is 48 bits, a maximum history of 48 states can be recorded. When an option control card appears that has a standard option name and an implicit option action, the resultant action is identical to that which would have reseulted had all 48 bits of each standard option stack been **RESET** and followed by an explicit **SET** performed on each indicated option. For example, after the appearance of an option control card containing:

```
$ SINGLE
```

the history stack for the **SINGLE** option contains a 1 in the top stack position and all zeros in the following positions. The history stack for each of the other compiler options would then contain all zeros. A compiler control statement that applies to compiler options begins with an explicit or implicit option action and contains a list of options to which the option action is to apply. This statement ends when the next implicit option action is encountered on the compiler control card or when a percent sign is encountered on the compiler control card or when a percent sign is encountered or column 72 of the card is reached. The compiler options affected by the compiler control card retain the indicated states for all input cards with sequence numbers greater than the sequence number on the compiler control card that has the control statement, or the physically succeeding input cards for a deck in which all sequence numbers are blank, until another compiler control card is encountered that alters the option states. The following illustration (figure D-1) is an example of a card that has compiler control statements employing option actions:

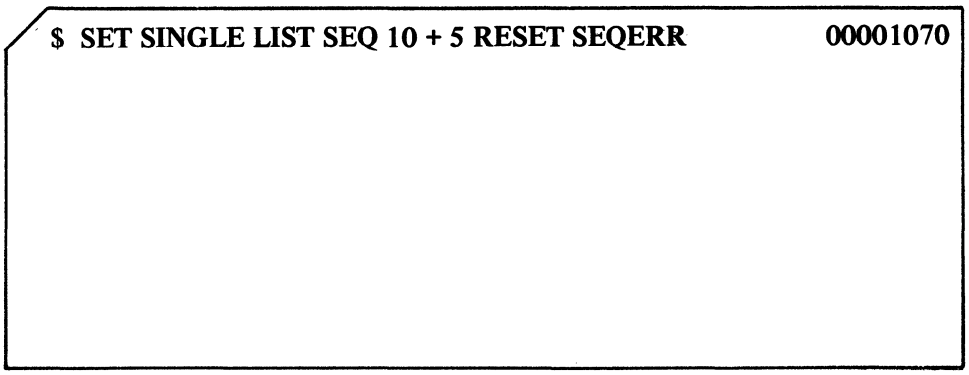


Figure D-1. Option Control Card

The option control card assigns the state **SET** to the options **SINGLE, LIST,** and **SEQ,** with the sequencing arguments of 10 and +5. It also assigns the state **RESET** to the option **SEQERR.** The card has the sequence number 00001070 in columns 73 through 80.

If an option control card is empty, it has no effect on other options; however, if there is a card image on the symbolic file with the same sequence number as the empty option control card, the image on the symbolic file is deleted.

OPTIONS

Standard Compiler Options

The compiler recognizes the following identifiers as standard compiler option names:

AUXLOGIC
CODE
DCPLIST
DUMP
ERRLIST
LIMIT
LIST
LISTOMITTED
LISTP
MERGE
NDLMACRO
NETWORK
NEW
NEWSEQERR
NOWARN
OMIT
PAGE
SEQ
SEQERR
SINGLE
SUMMARY
SYNTAX
VERSION
VOID
VOIDT
$

The standard options are discussed alphabetically in the following paragraphs. The default state of each option is indicated in parentheses following the option name; the function performed by the option is discussed in the paragraph accompanying the same.

AUXLOGIC (RESET)

All source code encountered when **AUXLOGIC** is true will result in the corresponding object DCP code being allocated in main memory for each **DCP** which utilizes the affected logic. This option might be used around error handling code, for example, or other infrequently-used logic.

Example:

```
                INITIATE TRANSMIT.
                TRANSMIT CHAR [BREAK:1C0].
                TERMINATE NORMAL.
$ SET           AUXLOGIC
100:            TOG[0]=TRUE.
                INITIALIZE RETRY.
101:            TERMINATE ERROR.
$ POP           AUXLOGIC
```

CODE (RESET)

The code option causes the printout to contain the compiler-generated object code.

DCPLIST (RESET)

If **SET**, lists code addresses of each source statement on the **LINE** file. (There will be two separate lists addresses if used for two **DCP**s, three for three **DCP**s, etc.)

DUMP (RESET)

If **SET**, causes a "raw dump" listing of **NIF** on the **LINE** file.

ERRLIST (RESET)

The **ERRLIST** option causes syntax error information for CANDE to be written on the **ERRORFILE** file. When a compilation error is detected in the source input, an error message is written in the **ERRORFILE** file. This option is provided primarily for use when the compiler is called from a remote terminal by the CANDE language, but it can be used regardless of the manner in which the compiler is called. When the compiler is called from CANDE, the default state of the **ERRLIST** option is **SET** and **ERRORFILE** is automatically equated to the remote device involved.

LIMIT (cannot be **SET** or **RESET**)

The integer parameter allows the user to control compiler error terminations. The proper format for the **LIMIT** option is as follows:

LIMIT ⟨*integer*⟩

Compilation is terminated if the number of errors detected by the compiler equals or exceeds the ⟨*integer*⟩. If no **LIMIT** statement appears, a default error limit of 150 is assigned unless the compilation is initiated through **CANDE**, in which case the default error is 10.

LIST (SET; RESET for CANDE)

The **LIST** option causes a printout to be generated on the compiler output **LINE** file. The contents of such printouts are specified in the preceding paragraphs describing compiler features. If the **LIST** option is **RESET**, only syntax error messages and compilation information are listed.

LISTOMITTED (defaults to setting of **LIST**)

The **LISTOMITTED** option causes records voided by the **OMIT** option to appear on the **LINE** file with "**OMIT**" displayed at the right margin.

LISTP (RESET)

When **SET**, the **LISTP** option causes patches and input records from the compiler **CARD** file to be included on the printout while records from the compiler **TAPE** file are excluded. This option is effective

only if the **LIST** option is **RESET**. If the **LIST** option is **SET**, the state of **LISTP** is ignored. Therefore, the **LISTP** or the **LIST** option causes a printout to be generated when **SET**.

MERGE (RESET)

When **SET**, the **MERGE** compiler option causes primary input, **CARD** file, to be merged with secondary input, **TAPE** file, to form the total input to the compiler. If matching sequence numbers occur, the primary input overrides. If the **MERGE** option is **RESET**, only primary input is used and secondary input is totally ignored. Therefore, the total input to the compiler when the **MERGE** option is **SET** consists of all card images from the **CARD** file, and all card images from the **TAPE** file that do not have sequence numbers that can be found on cards in the **CARD** file.

NDLMACRO (RESET)

If **SET**, the **NDL MACRO** interface code will be printed following each statement within a ⟨*request definition*⟩ or ⟨*control definition*⟩.

NETWORK (RESET)

When **SET**, the compiler option **NETWORK** will invoke a brief tabular Network Status Report. A report will be generated even if the dollar option **SYNTAX** is set or if the dollar option **LIST** is **RESET**.

The report contains the following fields and information:

- DCP number and exchange status
- Cluster number
- LINE information
  - Address (DCP number: cluster number: LINE number)
  - STATION(s) information
  - Special characters or address characters if specified
  - LSN
  - Terminal name (17 characters)
- Connection medium (direct or modem name, 17 characters)
- LINE speed, character size, adapter mode
- MCS Name (30 characters)

Examples:

```
$SET LIST SYNTAX NETWORK
$SET NETWORK
```

NEW (RESET)

When the state of the **NEW** option is **SET**, the merged input from the **CARD** and **TAPE** files is placed on the updated symbolic output file **NEWTAPE**. This file is coded in **EBCDIC** and is structured in 15-word records and 450-word blocks. Therefore, it can later be used as input to the compiler through the **TAPE** file. All option control cards in the merged **CARD** and **TAPE** file input are placed on the **NEWTAPE** file when **NEW** is **SET** and only if the initial $ sign on these cards is in card column 2.

The **NEW** option can be **SET** and **RESET** as necessary by option control cards appearing at any point in the input file. Such option control cards can also be placed on the **NEWTAPE** file if the $ signs on these cards are in column 2.

The **NEWTAPE** file is created despite the occurrence of syntax errors in the source input. This file can be used as a secondary input for a later compila ion.

The **NEWTAPE** file can be label-equated so that, for example, the output goes to magnetic tape.

NEWSEQERR (RESET)

The **NEWSEQERR** option causes sequence errors on the **NEWTAPE** file to be flagged. If sequence errors occur and the **NEWSEQERR** option is **SET**, the **NEWTAPE** file is not locked, and the message **NEWTAPE NOT LOCKED** {*number of errors*} **NEWTAPE SEQUENCE ERRORS** is printed on the printout. **NEWTAPE, NIF,** and **DCPCODE** files are not locked.

NOWARN (RESET)

When **SET**, suppresses any compiler warnings from appearing on the **LINE** file.

OMIT (RESET)

The **OMIT** option voids input from both the **CARD** and **TAPE** files until it is **RESET** or **POP**ped into a **RESET** state.

PAGE (cannot be **SET** or **RESET**)

The **PAGE** compiler option must appear on a option card without an option action preceding it. When a **PAGE** option card appears, the printout is spaced to the top of the next page, but only if the **LIST** option is **SET.**

SEQ (RESET)

The proper format of the **SEQ** option is as follows:

SEQ ⟨*base*⟩ + ⟨*increment*⟩

If the **SEQ** option is **SET**, the printout and the new secondary source language file, **NEWTAPE**, contain new sequence numbers as defined by the ⟨*base*⟩ and ⟨*increment*⟩. If the ⟨*base*⟩ and ⟨*increment*⟩ are unspecified, a base of 0 and increment of 10 are assumed.

This option has effect only when the **LIST** and/or **NEW** options are also **SET**. The sequence numbers that appear on the card images in these files when the **SEQ** option is **RESET** are identical to the sequence numbers on the corresponding cards in the input file.

Example

```
$ SEQ 100 + 100                                    00005000
```

This compiler control card specifies that, when the state of the **SEQ** option is **SET**, sequencing begins with the sequence number 00000100 and proceeds in increments of 100.

SEQERR (RESET)

The **SEQERR** option causes sequence errors on the **TAPE** file to be flagged. If sequence errors occur and the **SEQERR** option is **SET**, **DCPCODE** and **NIF** files are not locked, and the message **CODE FILE NOT LOCKED** {number of errors} **TAPE SEQUENCE ERRORS** is printed on the printout.

SINGLE (RESET)

The **SINGLE** option causes the printout to be single-spaced. When the **SINGLE** option is **RESET**, the printout is double-spaced. (Note that double-spacing is default.)

SUMMARY (RESET)

If **SET**, lists on the **LINE** file the memory space allocations for user translation tables and terminal message space allocations for each **DCP**.

SYNTAX (RESET)

When **SET**, the source program is checked for syntax errors only. **DCPCODE** and **NIF** files are not generated.

VERSION (SET, RESET, and POP are ignored by the compiler)

The **VERSION** compiler option allows the user to specify an initial version number for a source program, to replace an existing version number, or to append an existing version number.

Examples

```
$ VERSION 25.010.010
$ VERSION +01.+001.010
```

When compiling with the **NEW** compiler option **SET** and a **VERSION** compiler card appears in the symbolic, and if the patch deck contains a **VERSION** compiler option, the new symbolic is updated to the version,

cycle, and patch number on the last **VERSION** compiler card in the patch deck. The sequence number must be less than the one in the symbolic.

VOID (RESET)

If the **VOID** option is **SET**, all input **from the TAPE** and the **CARD** files is ignored by the compiler until the **VOID** option is **RESET** or **POP**ped into a **RESET** state. The ignored input is neither listed nor included in the updated symbolic file regardless of the states of the **LIST** and **NEW** options. The **VOID** option can be **RESET**, once it is **SET**, only by a option control card in the **CARD** file.

VOIDT (RESET)

If the **VOIDT** option is **SET**, all secondary input from the **TAPE** file is ignored by the compiler until the **VOIDT** option is **RESET** or **POP**ped into a **RESET** state. Therefore, while the **VOIDT** option is **SET**, only primary input is compiled. The ignored input is neither listed nor included in the updated symbolic file regardless of the states of the **LIST** and **NEW** options. The **VOIDT** option can be **RESET**, once it is **SET**, only by an option control card in the **CARD**.

$ (RESET)

When **SET**, the dollar sign ($) option causes the printout of all subsequent ⟨*option control card*⟩ images when the **LIST** option is **SET**. This option appears as **$SET $** or **$ $**.

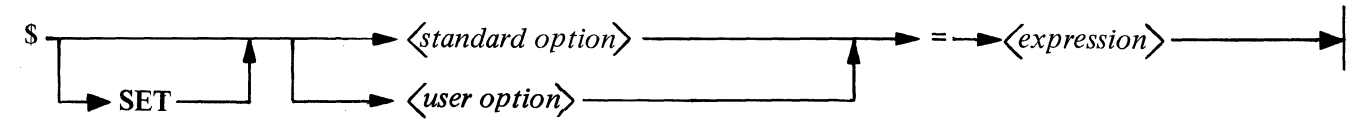
⟨*user defined option*⟩ **(RESET)**

A word on a option control card, which is not a standard option, is recognized as a user-defined option. It can be **SET**, **RESET**, or **POP**ped.

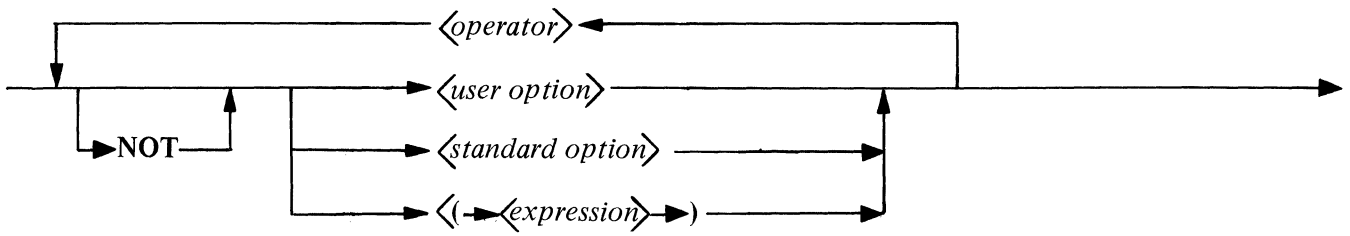
The first five characters and the length of user-defined options are significant. For example, the compiler control statements **"$SET ABCDEX"** and **"$RESET ABCDEY"** are treated as references to the same option, but **"$SET ABCDEXY"** is treated as a reference to a separate user option.

Option Expressions

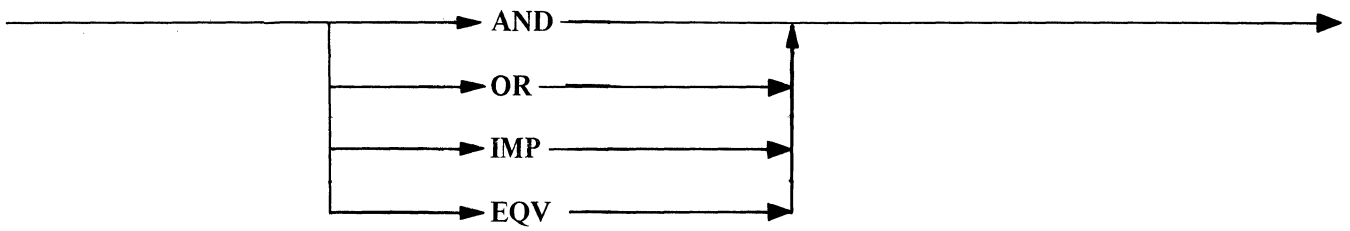
Syntax



⟨*expression*⟩



⟨*operator*⟩



Examples

```
$SET MYOPTION = MYOPTION OR NOT LIST
$SET LIST = NOT (LISTP OR MYOPTION) AND VOIDT
```

Semantics

Any option, standard or user, can be **SET** or **RESET** by "equating" it to an option expressions.

The ⟨*expression*⟩ on the right of the equal sign is evaluated as either **TRUE (SET)** or **FALSE (RESET)** by treating the options in the ⟨*expression*⟩ in the sense of logical variables. Depending on the results of the evaluation, the ⟨*option*⟩ on the left of the equal sign is either **SET** or **RESET**.

The logical operators are defined by Table D-1.

Table D-1. Truth Table

OPTION A	OPTION B	NOT A	A AND B	A OR B	A IMP B	A EQV B
SET	SET	RESET	SET	SET	SET	SET
SET	RESET	RESET	RESET	SET	RESET	RESET
RESET	SET	SET	RESET	SET	SET	RESET
RESET	RESET	SET	RESET	RESET	SET	SET

The precedence of the logical operators is

First: **NOT**
Second: **AND**
Third: **OR**
Fourth: **IMP**
Fifth: **EQV**

The order of precedence may be changed by using parentheses.

APPENDIX E

COMPILER SOURCE AND OBJECT FILES

COMPILER FILES

Compiler communication is handled through various input and output files (figure E-1). Cards, disk, or magnetic tape can be specified as source language input media. Input must be in the input format defined in the preceding sections. The compiler has the capability of merging, on the basis of sequence numbers, input from cards, tape, or disk. When inputs are being merged, indications of text insertions or replacements can be made to appear on the printout. In addition to the printout, the compiler can also generate updated symbolic files. These files can be created in addition to the compiler-generated output code file.

Input Files

The primary compiler input file is a card file with the internal name **CARD**; the secondary input file is a serial disk file with the internal name **TAPE**. The presence of the primary file **(CARD)** is required for each compilation; the presence of the secondary file **(TAPE)** is optional for each compilation. When two card images, one from the **CARD** file and the other the **TAPE** file have the same sequence number, the former is primary and is compiled, and the latter is ignored. This is the standard mode of handling source language input. File **CARD** can be either **BCL**-coded with 10-word records or **EBCDIC**-coded with 14-word records and can be either blocked or unblocked. File **TAPE** can be **BCL**-coded with 10-word records and 150-word blocks, or **EBCDIC**-coded with a 14- or 15-word record and 420- or 450-word blocks. Both the **CARD** file and the **TAPE** file can be label-equated (via the **FILE** system control card) to change the **TITLE** and **KIND** of the file. The **TAPE** file is used as input only when the **MERGE** compiler option is **SET**.

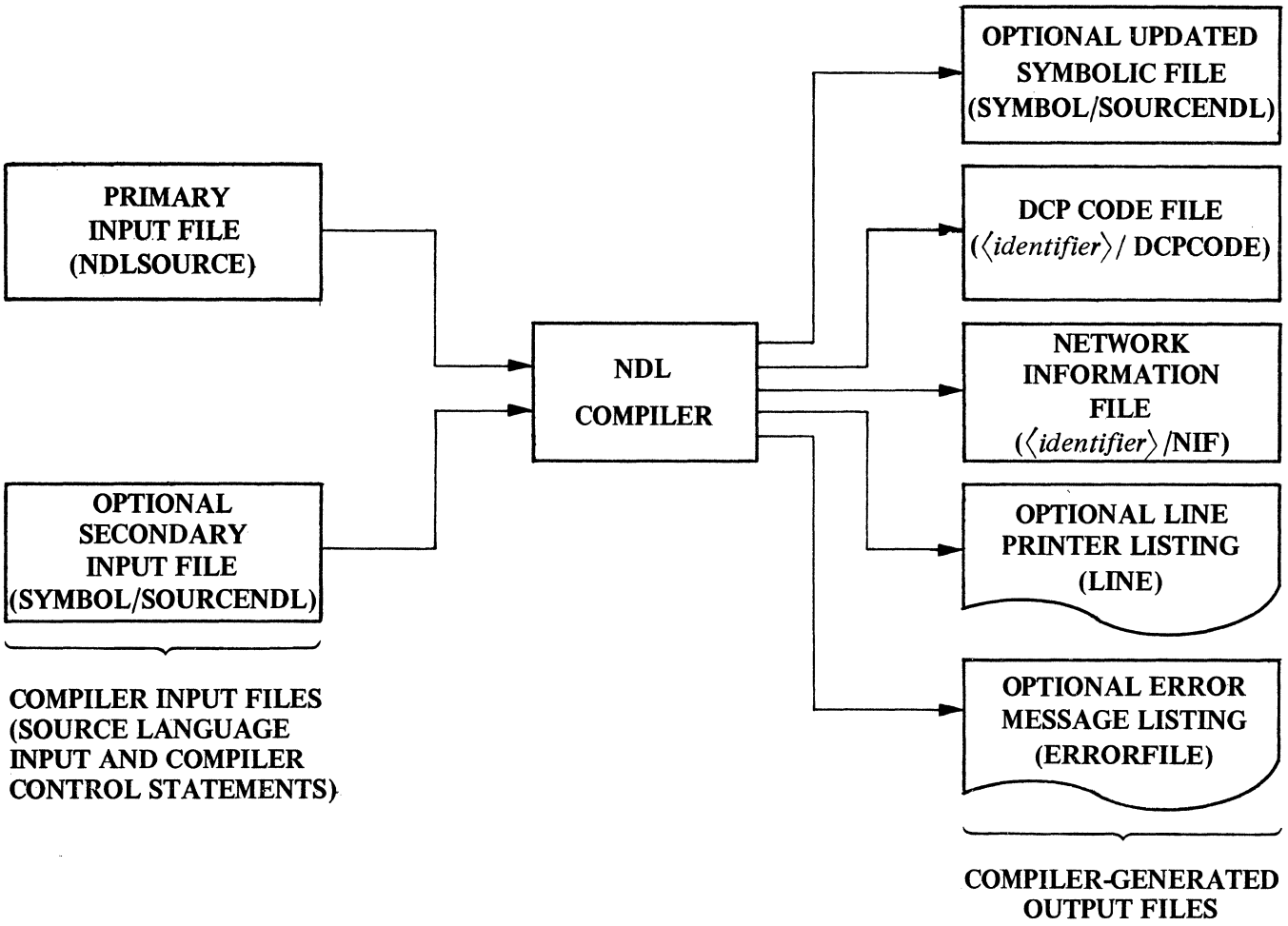


Figure E-1. **NDL** Compilation System

Output Files

Output files produced by the compiler consist of the DCP code file, the Network Information File, an updated symbolic file, a line printer printout, and an error message file. The **DCP** code file has the internal name **DCPCODE** and is saved on disk after the compilation unless the **COMPILE** system control card specifies compilation for syntax only, or unless syntax errors are detected in the source language input by the compiler. If compilation for library is specified, then the **DCPCODE** and **NIF** files are saved on disk. The title of the saved **DCP** code file is identical to the program name ⟨*identifier*⟩ appearing on the **COMPILE** system control card with the suffix of /**DCPCODE.**

The title of the saved Network Information File is identical to the program name ⟨*identifier*⟩ appearing on the **COMPILE** system control card with the suffix of /**NIF.**

The updated symbolic file is, by default, a disk file generated only if the compiler option **NEW** is **SET.** This file contains the compilation source input or a selected portion of this input as specified by the state of the **NEW** compiler option. It can be used as the **TAPE** file for a succeeding compilation. This output file has the internal file name **NEWTAPE** and contains EBCDIC-coded 15-word records in 450-word blocks.

The line printer printout is an optional print file that is created unless the compiler option **LIST** is **RESET**. (The **LIST** option is **SET** by default unless the compilation is initiated through CANDE.) The file has the internal name **LINE**, consists of 22-word EBCDIC-coded records, and contains the following information:

a. Source and compiler control statements used as input to the compiler.
b. Error messages and error count.
c. Number of input card images scanned.
d. Elapsed compilation time.
e. Processing time required for compilation.
f. Total number of words of DCP code generated.
g. Number of disk segments required for the DCP code file.
h. Title of the generated code file.

The NDL compiler uses the **TIME** (23) intrinsic to determine the proper machine identifier for the heading of compiler listings.

Depending upon the specified setting of the **LIST** and **CODE** compiler options, the line printer printout can contain more (or less) information than the basic items listed above. Card images from the **CARD** file are denoted on the printout by a **C** after the card contents. Card images from the **TAPE** file are denoted by a **T** in this location. A **P** denotes a patch of a **TAPE** card image.

The output error-message file with the internal file name and assigned title of **ERRORFILE** is an optional line printer file that is created when the **ERRLIST** compiler option is **SET**. This file is normally employed for compilations initiated through CANDE, in which case **ERRLIST** is **SET** by default and the **ERRORFILE** file is assigned to the remote device involved. The **ERRORFILE** file can also be used for compilations initiated through the card reader. This file is assigned EBCDIC-coded 12-word records that result in a line width of 72 characters, allowing the file to be used as output to a remote terminal or card punch without truncation of text. When a syntax error is detected, an error message is written following the line of text. The error message consists of an explanatory message and indicates the probable cause of the error.

Compiler File Table

Table E-1, NDL Compiler Files, lists the external name of the file (i.e., the name one would label-equate to), the internal name of the file (i.e., the name used when the file is declared within the compiler), the purpose served by the file, the default **KIND** of the file, the code used to store file data, the default record size (**MAXRECSIZE**) and block size (**BLOCKSIZE**) of the file, and a brief commentary on the specific file. The attributes of any of these files can be changed by the use of **FILE** system control cards directed to the compiler.

Table E-1. NDL Compiler Files

EXTERNAL NAME	INTERNAL NAME	PURPOSE	KIND	CODE	RECORD SIZE	BLOCK SIZE	COMMENTS
NDLSOURCE	**CARD**	Input Card File	**CARD READER**	**EBCDIC** **BCL**	14 Words 10 Words	Blocked or Unblocked	Required for each compilation. Primary compiler input file; may be label-equated to change file attributes. **CANDE** file is equated to this file automatically for compilations initiated through **CANDE**. Default title is **NDLSOURCE.** **BUFFERS = 2.** **FILETYPE = 8.**
SYMBOL/SOURCENDL	**TAPE**	Input Disk File	**DISK**	**EBCDIC** **BCL**	14 or 15 Words 10 Words	420 or 450 Words 150 Words	Optional file; need not be present for each compilation. Secondary compiler input file; selected as input by **SET**ting **MERGE** compiler option. Can be label-equated to change file attributes as desired. The default title is **SYMBOL/SOURCENDL.** **FILETYPE = 8.**
⟨*identifier*⟩**/DCPCODE**	**DCPCODE**	**DCP** Code File	**DISK**	Hexadecimal	30 Words	420 Words	Generated **DCP** code file. Saved or discarded and assigned a title as indicated by compilation method. For **CANDE** compilations, the title becomes: **OBJECT/**⟨*identifier*⟩**/DCPCODE.**
SYMBOL/SOURCENDL	**NEWTAPE**	Updated Symbolic Output File	**DISK**	**EBCDIC**	15 Words	450 Words	Optional output file produced when **NEW** compiler option is **SET.** This file contains portions of the source input and is label-equatable. It is suitable for use as a **TAPE** file for a later compilation. **BUFFERS = 2.** **AREASIZE = 1000.** **AREAS = 20.**

Table E-1. NDL Compiler Files (Cont)

EXTERNAL NAME	INTERNAL NAME	PURPOSE	KIND	CODE	RECORD SIZE	BLOCK SIZE	COMMENTS
⟨*identifier*⟩**/NIF**	**NIF**	Network Information File	**DISK**		30 Words	420 Words	Generated Network Information File **(NIF).** Saved or discarded and assigned a title as indicated by compilation method. For **CANDE** compilations the title becomes: **OBJECT/**⟨*identifier*⟩**/NIF.**
LINE	**LINE**	Line Printer Printout	**LINE PRINTER** or **REMOTE**	**EBCDIC**	22 Words	22 Words	Optional and label-equatable file. Produced when the compiler option **LIST** is **SET.**
ERRORFILE	**ERROR-FILE**	Error Listing Output File	**LINE PRINTER**	**EBCDIC**	12 Words	12 Words	Optional error listing file produced when **ERRLIST** compiler is **SET.** Contains card images and error messages. Automatically provided for **CANDE** input.

APPENDIX F

FULL DUPLEX

Full Duplex describes some portion of a data transmission link as capable of simultaneous transmission and reception of units of information. Half Duplex is capable of transmission or reception, but not simultaneously.

In order to have full-duplex capabilities, the following three functional areas are affected:

1. The cluster interface to a full-duplex line (or its modem) is effected by a Y-cable, connecting the line or modem to two line adapters, which assigns byte transmission leads to one adapter, byte reception to the other.

2. Two simultaneous NDL Requests or Control Logics (one for each adapter) are necessary to run for the same full-duplex line. Although parallel, byte transmissions and receptions on a full-duplex line are usually not completely independent. The two parallel, simultaneous, NDL logics must then be able to interact with one another. The NDL/DCP facility helps this interaction in several ways:

   (a) Both logics will address the same table space. NDL **TALLYS**, **TOGS**, etc., may then contain inter-logic control information.

   (b) Either adapter's NDL logic may control or be controlled by the other.

3. One of the adapters, and its NDL logic, is designated as Primary. As such, it controls the station queue. The other is designated as Auxiliary. The Auxiliary may control a single message space, not queued in the station queue. In this way, each NDL logic may have a separate message space associated with it, with an independent, parallel path to the MCP/DCP result queue, enabling full-duplex message transfers.

DATA SPACE

The network associated with each DCP or cluster exchange is defined in memory by a two-dimensional dope-vector like table structure:

1. The line vector is indexed by Cluster/Adapter address; this yields a line descriptor which contains the absolute address of the line table.

2. The line table consists of several line-associated information words followed by a station vector for the stations on the line. The station vector is indexed by the station number and yields a station descriptor which contains the absolute address of the station table.

3. The station table contains station-associated information.

Both primary and auxiliary lines have separate, valid line tables. Line tables are expanded to contain a line reference to an associated line. That is, the line table of the primary line will contain the Cluster/Adapter address of the auxiliary and vice versa.

The auxiliary line table contains its own independent reference to a line control routine, which may be different than the primary line control; adapter type, delays and timeout values; and its own line **TALLY**s and **TOG**s (in NDL these will be syntactically distinguished from those of the primary by the NDL reserved

word **AUXILIARY** or **AUX** used as a prefixed qualifier). The last characteristic expands the NDL controlled data space, as either set of **TALLY**s and/or **TOG**s may be referenced by NDL statements running for either line.

Of the line status flags (e.g., **WRITEREADY, ACKNOWLEDGEREADY, NOT READY, SWITCH STATUS**) only auxiliary line busy, i.e., **AUX(LINE(BUSY))**, is valid as a separate entity. All others are descriptive of the line pair and are valid only in the primary line table.

The station index is valid only in the primary table; the station vector is appended only to the primary. NDL statements executed for the auxiliary will reference station related quantities through the primary table structure.

NDL STATEMENTS

The following NDL statements are useful in programming full-duplex capabilities.

FORK Statement

The ⟨*fork statement*⟩ causes a second line to execute code beginning at a specified label, while the first line continues executing in parallel.

WAIT Statement

The ⟨*wait statement*⟩ causes a line to be temporarily suspended. It can be used for either the primary or auxiliary lines.

RECEIVE Statement

The **RECEIVE** ⟨*action part*⟩ includes a **CONTINUE** as a **RECEIVE** action.

Example:

```
RECEIVE(1 SEC) CHARACTER[ERROR[0], CONTINUE:3]
```

Such a ⟨*receive statement*⟩ is as a compound ⟨*receive statement*⟩ and ⟨*wait statement*⟩, allowing a line algorithm to respond to:

1. A byte received or any related error occurrence, such as a parity error or loss of carrier;
2. A timeout; or
3. Being continued by the other line.

CONTINUE Statement

The ⟨*continue statement*⟩ causes the other line to resume processing, provided it had been suspended by a ⟨*wait statement*⟩ or ⟨*receive statement*⟩ with a ⟨*continue action*⟩ specified; otherwise, the statement has no effect on the other line. In either case, the line on which the ⟨*continue statement*⟩ was executed itself continues without interruption.

ADAPTER TYPE

⟨*adapter type*⟩ allows a pair of adapter type values, for use in the modem, terminal and station declarations. This allows the specification of different adapter types for the primary and auxiliary lines. Refer to the individual descriptions of **MODEM**, **STATION** and **TERMINAL** ⟨*adapter statement*⟩s.

CONTROL Statement

The ⟨*control statement*⟩ in the **TERMINAL** section allows specification of two control routines. If the second control routine (for the auxiliary) is not specified for a full-duplex terminal, then the default equivalent of an ⟨*idle statement*⟩ will be used.

TERMINATE Statement

The ⟨*terminate statement*⟩ allows a simple, unqualified **TERMINATE**. As is true of all ⟨*terminate statement*⟩s, it may be used only in a **REQUEST**. The simple **TERMINATE** will serve only to exit from the **REQUEST** to the appropriate **CONTROL** routine (primary or auxiliary). The exit will be done without suspension of the line, with no effect on the ready status of the station and without returning or discarding any associated message space.

LINE TYPE Statement

The ⟨*line type statement*⟩ may include the clause ⟨*DUPLEX:line identifier*⟩.

Such a ⟨*type statement*⟩ is to be used in the line declaration for the primary line, and references the auxiliary line. The line referenced as auxiliary may itself not have a ⟨*type statement*⟩, nor may it have any stations.

INDEX

INDEX (CONT.)

Documentation Evaluation Form

Title: B 6000/B 7000 Systems NDL Reference Manual (Rel. Level 3.5)

Form No: 1176690

Date: June, 1985

Burroughs Corporation is interested in receiving your comments and suggestions regarding this manual. Comments will be utilized in ensuing revisions to improve this manual.

Please check type of Comment/Suggestion:

☐ Addition ☐ Deletion ☐ Revision ☐ Error ☐ Other

Comments:

From:

Name _____

Title _____

Company _____

Address _____

_____

Phone Number _____ Date _____

Remove form and mail to:

Burroughs Corporation
Corporate Product
Information East
209 W. Lancaster Ave.
Paoli, PA 19301 U.S.A.