


Burroughs 

**Computer Management System
(CMS)**

**Network Definition
Language (NDL)**

REFERENCE MANUAL

PRICED ITEM

**Computer Management System
(CMS)**

**Network Definition
Language (NDL)**

REFERENCE MANUAL

Copyright © 1980 Burroughs Corporation, Detroit, Michigan 48232

PRICED ITEM

"The names used in this publication are not of individuals living or otherwise. Any similarity or likeness of the names used in this publication with the names of any individuals, living or otherwise, is purely coincidental and not intentional."

Burroughs believes that the software described in this manual is accurate and reliable, and much care has been taken in its preparation. However, no responsibility, financial or otherwise, can be accepted for any consequences arising out of the use of this material, including loss of profit, indirect, special, or consequential damages. There are no warranties which extend beyond the program specification.

The Customer should exercise care to assure that use of the software will be in full compliance with laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change. Revisions may be issued from time to time to advise of changes and/or additions.

This edition includes the information released under the following:

PCN 1090925-001 (November 30, 1977)

PCN 1090925-002 (October 30, 1978)

Correspondence regarding this document should be addressed directly to Burroughs Corporation, Technical Information Organization, Software Systems Support, P. O. Box 235, Downingtown, Pennsylvania 19335.

TABLE OF CONTENTS

Section	Title	Page	Section	Title	Page
	INTRODUCTION	viii		IIR Register	2-11
1	THE NETWORK DEFINITION			LCHAR Register	2-11
	LANGUAGE	1-1		TOGGLES	2-11
	General	1-1		Byte-Variable Bit Reference	2-11
	NDL Concept	1-1		CRC Toggle	2-12
	Constant Section	1-1		Error Flags	2-12
	Translation Section	1-2		Item Error Flags	2-13
	Line Control Section	1-2		BCCERR	2-13
	Request Section	1-2		ADDERR	2-13
	Modem Section	1-2		TRANERR	2-13
	Terminal Section	1-2		ENDOFBUFFER	2-13
	Station Section	1-2		CRCERR	2-13
	Line Section	1-2		FORMATTER	2-13
	DCP Section	1-2		PARITY	2-13
	File Section	1-2		STOPBIT	2-13
	NDL Line and Station Tables	1-3		TIMEOUT	2-13
	NDL Program Structure	1-3		BREAK	2-14
2	NDL LANGUAGE ELEMENTS	2-1		BUFOVFL	2-14
	General	2-1		LOSSOFCARRIER	2-14
	Character Set	2-1		ABORT	2-14
	Notation Description	2-1		IDLE	2-14
	Key Words	2-1		INVALID	2-14
	Lower Case Words	2-2		Break Error Flags	2-14
	Braces	2-2		BREAK[RECEIVE]	2-14
	Consecutive Periods	2-2		BREAK[TRANSMIT]	2-14
	Brackets	2-2		LINE Toggles	2-14
	Parentheses	2-2		AUXILIARY LINE Toggles	2-15
	Punctuation	2-3		LINE STATUS Toggles	2-15
	Identifiers	2-3		BUSY	2-15
	Integers	2-3		AUX(LINE)STATUS Toggles	2-15
	Strings	2-3		QUEUED	2-15
	Character Strings	2-3		NORESPONSE Toggle	2-16
	Hexadecimal Strings	2-3		NOSPACE Toggle	2-16
	Unitary String	2-4		REQUEST Toggles	2-16
	Value	2-4		NAKONSELECT	2-16
	Reserved Words	2-4		NAKFLAG	2-16
	Labels	2-4		CONTROLFLAG	2-16
	Null and Empty Labels	2-4		WRUFLAG	2-16
	Byte-Variables or Special			BLOCK/BLOCKED	2-17
	Register Names	2-5		TRANSPARENT	2-17
	BCC/CRC Register	2-6		EVENT1	2-17
	CHARACTER Register	2-6		STATION STATUS Toggles	2-17
	LINE TALLY Register	2-7		READY	2-17
	AUXILIARY LINE Registers	2-7		QUEUED	2-17
	RETRY Register	2-7		ENABLED	2-17
	STATION FREQUENCY Register	2-8		VALID	2-17
	STATION Register	2-8		SYNCS Toggle	2-18
	TOGS Register	2-8		USER Toggle	2-18
	STATION TALLY Register	2-9		UNDERFLOW Toggle	2-19
	USER TALLY Registers	2-9		VARIANT Toggles	2-19
	MAXSTATIONS Register	2-10		SKIP	2-19
	SKIPCONTROL Register	2-10		PAGE	2-19
	IR Register	2-10		CARRIAGE	2-19

TABLE OF CONTENTS (CONT)

Section	Title	Page	Section	Title	Page
	LINEFEED	2-19		ADDRESS-Item	4-22
	PAPERMOTION	2-20		BCC-Item	4-22
	SPACE	2-20		STRING-Item	4-23
3	CONSTANT AND TRANSLATION SECTION	3-1		CRC-Item	4-23
	General	3-1		FRAMEADDR-Item	4-23
	Constant Section Description	3-1		RESTORE	4-25
	Translation Section Description	3-1		RETURN	4-25
	General	3-1		SHIFT	4-26
	Translation BYTE	3-3		STORE	4-26
	Translation SHIFT	3-3		STORE CHAR	4-27
	Translation DEFAULTCHAR	3-3		STORE String	4-27
	Translation TRANSLATE	3-4		SUM	4-27
4	REQUEST AND LINE CONTROL SECTIONS	4-1		TERMINATE	4-28
	General	4-1		TERMINATE NOINPUT	4-28
	General Line Control Section Description	4-1		TERMINATE NORMAL	4-28
	Line Control/Request Differences	4-1		TERMINATE ERROR	4-28
	Line Control Format	4-1		TERMINATE BLOCK	4-29
	General Request Section Description	4-1		TERMINATE ENABLEINPUT	4-29
	Request Section Format	4-2		TERMINATE	4-29
	ACU	4-3		TERMINATE NOLABEL	4-29
	ADDRESS	4-3		TERMINATE SAVE	4-29
	ASSIGNMENT	4-3	5	TERMINATE PRIORITYSAVE	4-29
	BACKSPACE	4-5		TIMER	4-29
	BINARY (Request)	4-6		TRANSMIT	4-30
	CONTINUE	4-6		WAIT	4-31
	DATASET	4-6		MODEM SECTION	5-1
	DELAY	4-6		General	5-1
	DELIVER	4-7		MODEM Section Description	5-1
	DISABLE	4-7		LOSSOFCARRIER	5-2
	DISCONNECT	4-8		NOISEDELAY	5-2
	ENABLE	4-8		SPEED	5-2
	ERROR	4-8	6	TRANSMITDELAY	5-3
	FETCH	4-10		TYPE	5-3
	FINISH TRANSMIT	4-10		ANSWERTONE	5-4
	FLAGFILL	4-11		TERMINAL SECTION	6-1
	FORK	4-11		General	6-1
	GETSPACE	4-12		TERMINAL Definition Description	6-1
	GO TO	4-13		ADDRESS	6-2
	IDLE	4-13		BACKSPACE	6-2
	IF	4-14		BLOCKED	6-2
	INCREMENT	4-16		BYTE	6-3
	INITIALIZE	4-16		CARRIAGE	6-3
	INITIATE	4-17		CLEAR	6-3
	LABEL	4-18		CODE	6-4
	PAUSE	4-18		CONTROL	6-4
	RECEIVE	4-19		DEFAULT	6-5
	CHARACTER-Item	4-20		END	6-5
	TEXT-Item	4-21		HOME	6-5
	TRAN-Item	4-22		LINEDELETE	6-6
				LINEFEED	6-6
				MAXINPUT	6-6
				MOD	6-6
				PAGE	6-7

TABLE OF CONTENTS (CONT)

Section	Title	Page	Section	Title	Page
	PARITY	6-7		MODEM	8-4
	REQUEST	6-8		TELEX	8-4
	SAVE	6-8		DIALIN	8-5
	SCREEN	6-9		DIALOUT	8-5
	SPEED	6-9		DUPLEX	8-5
	TALLIES	6-9		BITS	8-5
	TIMEOUT	6-10	9	DCP SECTION	9-1
	TRANSMISSION	6-10		General	9-1
	TRANSPARENT	6-10		DCP Definition Description	9-1
	TURNAROUND	6-11		BUFFERCOUNT	9-2
	TYPE	6-11		BUFFER	9-2
	WRAPAROUND	6-12		CODEFILE	9-2
	WIDTH	6-12		DCP MEMORY	9-2
	WRU	6-13		MEMORY	9-3
7	STATION SECTION	7-1		LIMIT	9-3
	General	7-1		TERMINAL	9-3
	STATION DEFAULT Description	7-1	10	FILE SECTION	10-1
	ADDRESS	7-2		General	10-1
	CONTROL	7-2		FAMILY	10-2
	DEFAULT	7-2			
	ENABLEINPUT	7-3	11	OPERATION OF CMS NDL	
	FREQUENCY	7-3		COMPILER	11-1
	INITIALIZE	7-3		General	11-1
	LOGIN	7-4		Hardware	11-1
	MODEM	7-4		Software	11-1
	MYUSE	7-4		Compiler Operation	11-1
	PAGE	7-5		Execute	11-1
	PHONE	7-5		Value Clause	11-1
	RETRY	7-5		File Equate Clause	11-2
	SPEED	7-6		Syntax	11-3
	SPO	7-6		Continuation	11-4
	TALLIES	7-7		Console Input	11-4
	TERMINAL	7-7		Source Merge Process	11-4
	TYPE	7-7		Library Generation and Usage	11-4
	WIDTH	7-8		Compiler Options	11-4
	WRAPAROUND	7-8		Option Format	11-4
8	LINE SECTION	8-1		Option Control	11-5
	General	8-1		Dollar Options	11-5
	LINE DEFINITION Description	8-1		Dollar LIST	11-5
	ADDRESS	8-2		LST1	11-5
	B 800 Systems	8-2		VOID NNNNNNNN	11-5
	B 800 and B 776 Systems	8-2		RSEQ NNNNNNNN +MMMM-	
	B 900 Systems	8-2		MMMM	11-5
	B 1800 Systems	8-2		CODE	11-5
	DEFAULT	8-2		MTCH	11-5
	MAXSTATIONS	8-3		SUPR	11-5
	MODEM	8-3		NPRT	11-5
	RATESELECT	8-3		SPEC	11-5
	STANDBY	8-4		LNXX	11-5
	STATION	8-4		CARD	11-6
	TYPE	8-4		DISK or DISC	11-6
	DIRECT	8-4		TAPE	11-6
	BDI	8-4		NEWC	11-6

TABLE OF CONTENTS (CONT)

Section	Title	Page	Section	Title	Page
	NEWD	11-6		Program Flow	13-10
	NEWT	11-6		Accessing the NDLSYS File	13-10
	LGEN	11-6		Generating Microcode	13-13
	LIBR	11-6		S-Instruction Processing	13-16
	TBLS	11-6		S-Instruction Processing	
	Comments	11-6		Examples	13-16
	Top of Page	11-6		Label Resolution	13-21
12	B 800 POST PROCESSOR	12-1		S-Label Generation	13-21
	PROGRAM	12-1		M-Label Resolution	13-21
	General	12-1		Control/Request Label	
	Post Processor Action	12-1		Resolution	13-22
	Execution of Post Processor			Line Discipline	13-23
	Program	12-1		Subroutines	13-23
	Data Comm Loader	12-1		Subroutine Generation	13-24
	Microcode Listing	12-1		Microcode Space Overflow	
13	NDL POST COMPILER	13-1		Detection	13-25
	Introduction	13-1		Types of Storing	13-25
	Overview	13-1		Microcode Storing	13-25
	Operating Procedures	13-1		Data Storing	13-26
	Executive NPC	13-3		Manager Schemes	13-26
	DBG	13-3		Multi-Line Manager	13-26
	NO.LIST	13-3		Idle	13-27
	FILE	13-3		Pause	13-27
	WORK	13-4		Delay	13-28
	STATUS	13-4		Receive	13-28
	Error Messages	13-4		Transmit	13-28
	Definitions	13-4		Full-Duplex Managers	13-28
	NDLSYS	13-4		Single-Line Manager	13-28
	S-CODE	13-4		Code Optimization	13-28
	S-OP	13-4		Terminal Types	13-28
	S-LABEL/S-ADDRESS	13-5		Terminal Code Reduction	13-29
	LABEL RESOLUTION	13-5		DCP-Related Optimization	13-30
	MANAGER SCHEME	13-5		Adapter	13-30
	HOST CONTROL	13-5		Registers	13-30
	POST COMPILER/NPC			NDL Compiler-Related	
	MEMORY	13-5		Optimization	13-30
	NPC Functions	13-5		Line Busy Information	13-31
	Program Structure	13-5		TRANSPARENT = TRUE/	
	Environment Establishment	13-5		FALSE	13-31
	Microcode Generation	13-7		TERMINATE BLOCK	13-31
	Microcode Listing Generation	13-7		SYNC = TRUE/FALSE	13-31
	File Structure	13-7		CRC = TRUE/FALSE	13-31
	Generated Microcode File			SHIFT = UP/DOWN/	
	(M.CODE.FILE)	13-7		MIDDLE	13-31
	Known S-Label Address File			STATION = LIS/VARIABLE	13-31
	(SLBL.K)	13-7		Debug Option	13-31
	Unknown S-Label Address			Printer File Listing	13-32
	File (SLBL.UN)	13-8			
	NDL Object Code File		A	NDL RESERVED WORD LIST	A-1
	(NDLSYS)	13-8	B	NETWORK SEMANTIC	
	Printer Backup File (PBM)	13-9		CONSIDERATIONS	B-1
	Printer Backup Key File			Terminal-Control/Request	B-1
	(PBM.KEY)	13-9		TERMINATE ENABLEINPUT	B-1

TABLE OF CONTENTS (CONT)

Section	Title	Page	Section	Title	Page
	TRANSMIT TEXT	B-1		MYUSE	B-2
	RECEIVE TEXT	B-1		ADDRESS	B-2
	GETSPACE	B-1		WIDTH	B-2
	STORE	B-1		PAGE	B-2
	TEXT CONTROL CHARACTERS	B-1		WRAPAROUND	B-2
	WRUFLAG	B-1		Line-Modem	B-2
	Station-Control/Request	B-1		SPEED	B-2
	CONTROL	B-1		DIALIN	B-2
	CONTROL FLAG	B-1		DIALOUT	B-3
	Station-Modem	B-1		DUPLEX	B-3
	TYPE	B-1		Line-Terminal	B-3
	SPEED	B-2		DUPLEX	B-3
	Station-Terminal	B-2		CONSISTENCY ALONG LINE	B-3
	TYPE	B-2		Line-Station	B-3
	TYPE OPTION	B-2		CONNECT TYPE	B-3
	SPEED	B-2		CONSISTENCY ALONG LINE	B-3
	STOPBITS	B-2	C	SAMPLE NDL PROGRAM	C-1

LIST OF ILLUSTRATIONS

Figure	Title	Page	Figure	Title	Page
13-1	NDL Post Compiler Flow	13-2	13-3	PBM File	
13-2	NDL Post Compiler Flow	13-6		Listing	13-33

LIST OF TABLES

Table	Title	Page	Table	Title	Page
1-1	The NDL Program	1-1	2-1	Character Set	2-1

INTRODUCTION

The Burroughs Network Definition Language (NDL) is a high-level language that enables a user programmer to efficiently define and implement a unique data communications environment.

The NDL source statements discussed in this manual are converted by the NDL compiler into the tables and codes needed for a custom data communications environment. The output of the compiler is used by the communications subsystem to perform the various line disciplines, buffer management, message queueing, automatic retries, and character translation.

In this manual, the various sections of NDL and the statements within each section are covered in detail. Examples are provided to give the programmer direct insight into the practical use of NDL in solving data communications problems.

SECTION 1

THE NETWORK DEFINITION LANGUAGE

GENERAL

This section provides a general description of the Network Definition Language (NDL). Succeeding sections explain in detail the information introduced in this section.

NDL CONCEPT

The NDL data communication environment is mainly a table-driven process. The written description in the NDL produces tables which the software uses to control the data communications network. An NDL program produces a description summarized by its various tables.

NDL has a subset of constructs which comprise a programming language. In this language, an NDL user programs (describes) the line discipline routines through which the communications subsystem communicates with the remote devices.

An NDL program is divided into sections which are arranged in the order given in Table 1-1.

Table 1-1. The NDL Program

Name of Section	Function
Constant	To equate names with strings for convenience in other descriptions.
Translation	Provides for user-defined translation tables.
Line Control	To define line control procedures.
Request	To define line disciplines used for remote devices.
Modem	To describe the modems used for lines.
Terminal	To describe physical hardware characteristics of remote devices.
Station	To describe logical characteristics and file characteristics for specified remote devices.
Line	To specify type of line and connected station(s).
DCP	To define the message space requirements for the communications subsystem.
File	To specify related sets of stations included as a file.

Constant Section

This section provides a method for equating identifiers with strings of characters. Constant identifiers may then be used outside of the Constant sections syntactically and semantically equivalent to the occurrence of that string named by the identifier.

Translation Section

This section provides the user with the capability to translate one character set (via user-supplied translation tables) to the internal character set ASCII. Translation is performed automatically by the system. This Translation section provides the user with the ability to define non-standard character sets.

Line Control Section

This section provides the logic that initiates line functions and specifies stations on a particular line. This line control logic is invoked in the communications subsystem operating system at several distinct times.

Line control logic is initiated whenever:

1. The line is not busy and an action is queued for the line.
2. A request is terminated.
3. A line is initialized.

Request Section

Within this section the user gives the line disciplines for the different types of terminals within the network to the data communications system. The program or description provided by the user is converted by the NDL compiler into S-language.

Modem Section

This section describes the modems that may be included with the data communications network. The important considerations in describing a modem are the devices it is compatible with and the timing considerations needed to calculate delays when "turning a line around". Compatibility depends upon the type of transmission (SYNC or ASYNC) and the speed of the lines to which the modem can be connected.

Terminal Section

This section describes the physical aspects of the terminal devices within the data communications network required by the data communications software to receive and transmit messages. Among the required features that must be included in a terminal definition are buffer size, character set (code), parity, and the requests (as specified within the Request section) to be used by the system to communicate with that type of device. It is important to point out that terminals are physical devices having, primarily, physical properties, while stations represent the individual remote elements in the system and are described by location and terminal type.

Station Section

This section provides the means to specify the logical characteristics of the remote elements within the data communications network. This section is used to specify the type of terminal a station is associated with and how it is used (such as input, output, or both). Also, each station has characteristics which make it unique within the system.

Line Section

This section provides the means to describe the logical and physical characteristics of the lines into the data communications system.

DCP Section

This section provides the means to define the message buffer size and the number of buffers to utilize message space in memory more efficiently.

File Section

This section provides the association of stations that are part of the network with the data communications file of an object job. The station referenced in the File section must be defined in the Station section.

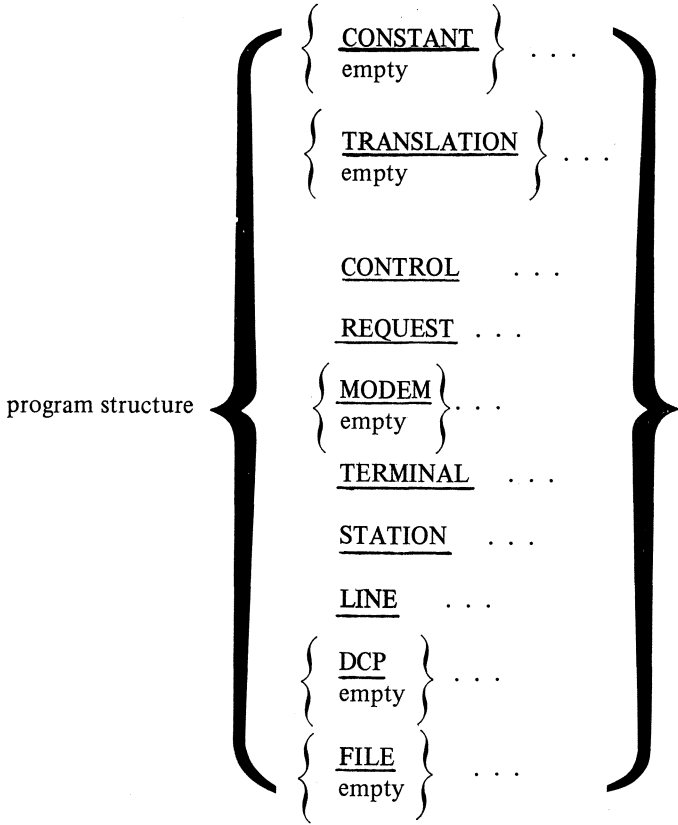
NDL LINE AND STATION TABLES

The communications subsystem software uses a set of line and station tables created by the NDL compiler and contained in the main storage.

The tables contain status information and parameters to modify or select communications subsystem logic. They also contain memory addresses that structure the table space to match the structured data communications network being controlled by the communications subsystem.

NDL PROGRAM STRUCTURE

The following represents the basic structure of an NDL program:



SECTION 2

NDL LANGUAGE ELEMENTS

GENERAL

NDL is a descriptive language and certain aspects must be previously stated so that the language can be understood. The following elements of NDL are used throughout this manual.

CHARACTER SET

The basic character set of NDL consists of 56 characters including letters, digits, and special characters. (See Table 2-1.)

Table 2-1. Character Set

O	—	9
A	—	Z
		blank or space
+		plus sign
-		minus sign or hyphen
*		asterisk
/		slash
=		equal sign
\$		dollar sign
,		comma
.		period or decimal point
;		semicolon
"		quotation mark
(left parenthesis
)		right parenthesis
>		greater than symbol
<		less than symbol
:		colon
@		at sign
%		percent sign
]		left bracket
[right bracket

NOTATION DESCRIPTION

The following notation conventions enable the reader to interpret the NDL syntax presented in this manual.

Key Words

All underscored, uppercase words are key words. Key words are required when the functions of which they are a part are used. The omission of key words causes an error condition during compilation. In the following example, the key words are IF, THEN, and GO TO.

IF toggle

THEN

GO TO label

Lower Case Words

All lower case words represent generic terms that must be supplied in that format position by the NDL programmer. In the following example, "break-time" and "break-delay-time" are generic terms.

$$\underline{\text{BREAK}} \quad \left\{ \begin{array}{l} \text{break-time,} \quad \text{break-delay-time} \\ \underline{\text{NULL}} \end{array} \right\}$$

Braces

When words or phrases are enclosed in braces, { }, a choice of one of the words or phrases is to be made. In the following example, one of the key words must be chosen.

$$\underline{\text{TIMEOUT}} = \text{integer} \quad \left\{ \begin{array}{l} \underline{\text{MILLI}} \\ \underline{\text{SEC}} \\ \underline{\text{MIN}} \end{array} \right\}$$

Generic terms may also be enclosed in braces and a choice of one is to be made.

$$\left\{ \begin{array}{l} \text{receive-error} \\ \text{item-error} \\ \text{transmit-error} \end{array} \right\}$$

When the lower case term "empty" appears with the braces, words and phrases enclosed in the braces represent optional portions of a statement. If the programmer wishes to include the optional feature, the entry shown between the braces may be included. Otherwise, it may be omitted. In the following example, the use of FETCH by itself is valid.

$$\underline{\text{FETCH}} \left\{ \begin{array}{l} \left[\left\{ \begin{array}{l} \underline{\text{ENDOFBUFFER:}} \\ \text{action-label} \\ \underline{\text{NULL}} \end{array} \right\} \left\{ \begin{array}{l} \text{action-label} \\ \underline{\text{NULL}} \end{array} \right\} \right] \\ \text{empty} \end{array} \right\}$$

Consecutive Periods

The presence of ellipsis (...) within any format indicates that the data immediately preceding the notation may be gone through, repeated, and another choice made, depending on the requirements of problem solving.

Brackets

Brackets, [], (when appearing in a statement's notation) must appear in the same position whenever the source program calls for the use of that statement.

Parentheses

Parentheses, (), appearing in a statement's notation must appear in the same position whenever the source program calls for the use of that statement.

Punctuation

Periods (.), commas (,), and equal signs (=) must appear in the same position in the source statement as they appear in the syntactical notation of that statement.

IDENTIFIERS

Identifiers are used to name constants, line control and request procedures, terminals, modems, stations, DCPs, translate tables, files, and lines. They have no intrinsic meanings.

An identifier may consist of letters and digits, but must begin with a letter. The maximum length of an identifier is 17 characters, of which the first 12 characters must be unique. characters. For file names and station names, the maximum length is 12 characters.

Valid Identifiers	Invalid Identifiers
ETX	ENQ-1
NULL	17LINE
TC500	9352
LN0175	TC500

INTEGERS

An integer consists of one or more digits. Only positive integers are allowed in NDL; spaces may not appear within an integer. They may be up to five digits in length with a maximum value of 65535.

Valid Integers	Invalid Integers
1	A1
123	1 6 2 6 3
255	2 1 5 C 0 9 8 5 7 5

STRINGS

A string (literal) is defined as an item composed of any allowable characters or hexadecimal values. The beginning and end of a string is denoted by a quotation mark. Any character enclosed within quotation marks is part of that string; subsequently, all spaces enclosed in the quotation marks are considered part of the string.

To represent a quotation mark in a string, two quotation marks in succession are used.

Character Strings

Simple character strings may be up to 248 characters in length and may contain any allowable character.

Examples

“YOU ARE NOW ON LINE”
“THIS IS TRANSMISSION NO. 1”

When characters are to be represented as hexadecimal strings, their ASCII representation must be used.

Hexadecimal Strings

A simple hexadecimal string may be up to 248 hexadecimal elements in length. A hexadecimal string must be preceded by a 4. If a hexadecimal string has an odd length, a single leading zero is assumed.

Section 2
NDL Language Elements

Examples

Valid Hex String	Invalid Hex String
4"05"	4"FIG2"
4"A305"	
4"A30"	

Unitary String

A unitary-string is defined as one character ("A") or two hexadecimal (4"03") elements.

VALUE

A value is either an identifier of a declared constant, an integer, or a unitary string (character or hexadecimal type) used as an operand in a CONTROL/REQUEST statement. Most values expressed in NDL compiler language are those which can be transformed by the NDL compiler into an eight-bit configuration because most of the value-receiving fields are that length. The only divergences from this rule are the STORE, RECEIVE, and TRANSMIT statements, which allow strings of multiple characters, and the toggle ASSIGNMENT statement.

Examples

Valid	Invalid
STATION = 40.	STATION = 300.
RETRY = 20.	RETRY = 4"FOFOFO".
RECEIVE [4"03":22].	RECEIVE ["STOP":34].
TOG [7] = TRUE.	TOG [7] = 1.

RESERVED WORDS

Appendix A provides a list of reserve words.

LABELS

Statements in the Line Control and Request sections are executed sequentially as listed. However, when it is desired to alter the order of execution of statements, the statement to which control is to transfer must be designated. A statement is designated by assigning a label to that statement.

A label may consist of up to five digits followed by a colon and then followed by the statement associated with that label.

Examples

Valid Labels	Invalid Labels
01:	1A:
12345:	B1:
1:	1 23456:

Null and Empty Labels

An empty label within a statement causes control to be passed to the next sequential instruction. A NULL label also does this, except when used in a RECEIVE or TRANSMIT TEXT instruction. In that case, control remains in that statement.

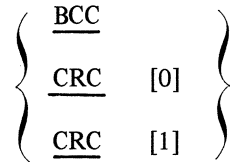
BYTE-VARIABLES OR SPECIAL REGISTER NAMES

The NDL provides various registers that may be utilized by the NDL programmer to aid in the control of the data communications environment. Some of these are read only registers (value may not be changed by the NDL programmer), while others may be used on either side of the equal (=) sign in the ASSIGNMENT statement. The byte-variable names are:

BCC	RETRY
CHAR	STATION
LINE(TALLY[0 or 1])	STATION(FREQUENCY)
AUX(LINE(TALLY [0 or 1]))	TALLY [0-18]
MAXSTATIONS	TOGS
IR	SKIPCONTROL
IIR	STATION(TALLY)
LCHAR	
CRC[0]	
CRC [1]	

BCC/CRC

This register is a work area used to collect a value derived from the technique specified in the PARITY statement (Terminal section), the INITIATE and SUM statements (Line Control and Request section), ASSIGNMENT of the CRC toggle (Control/Request), and the choices of the NDL programmer with respect to the SUM statement. Any RECEIVE, TRANS CHAR or TEXT updates the value in this register.



BCC/CRC may be used in control and request sets and may only be interrogated (IF statement) for a specific value. BCC/CRC may appear only on the right side of an ASSIGNMENT statement (c.f., INITIALIZE, SUM).

The CRC register is 16 bits in length in contrast to the eight-bit length of the BCC value. Syntactically, CRC is a toggle. To resolve ambiguity of toggle-versus-register reference, the left-most eight bits of CRC [1] and the remainder as CRC [0] are shown in the examples below.

Examples

```
CRC [1] = CRC [0].  
IF CRC [0] GTR 4 "OF" THEN CRC [1] = 1.
```

CHARACTER REGISTER

CHAR and CHARACTER are synonyms. The contents of this register may be tested for only a single value at a time. The register is loaded by an instruction such as the following:

```
RECEIVE CHAR.  
TRANSMIT TEXT.  
RECEIVE TEXT.
```

The character register contains the last character that was received or transmitted. Thus, upon the successful completion of TRANSMIT TEXT, the character register contains the text character transmitted last. Similarly, upon successful completion of RECEIVE TEXT, the character register contains the last character of the received text. These general rules are overridden in the following cases:

1. A FETCH statement causes the character register to be filled with the next character of the text buffer which has not already been fetched explicitly by a FETCH statement or implicitly by a TRANSMIT TEXT construct.
2. An explicit ASSIGNMENT statement such as CHARACTER = S causes a change in the contents of the character register.
3. If a TIMEOUT, LOSS OF BUFFER, or END OF BUFFER occurs, the character register contains the last previously received, transmitted, or loaded character.
4. If exit via an error such as STOPBIT, PARITY, or BYTE OVERFLOW occurs, the character register contains the last character received, untranslated.
5. If a BREAK occurs, the character register contains binary zeroes.
6. If a match was made on a specified literal (such as ETX), the character register contains that value.
7. Whenever an address function value is referenced in NDL, the contents of the character register is destroyed.
8. If an INVALID FRAME occurs, the character register contains a flag sequence. This is valid only for bit-oriented line disciplines.
9. If an exit via an error such as ABORT or IDLE occurs, the character register contains the abort sequence previously detected on the line. This is valid only for bit-oriented line disciplines.

The character register may be used in a logical expression or an ASSIGNMENT statement.

Examples

```
IF CHAR NEQ ETX THEN GO TO 10.  
CHARACTER = 4'03'.  
TALLY [1] = CHARACTER.
```

LINE TALLY REGISTERS

There are two of these registers associated with each line. They are general purpose registers and may be used by the NDL programmer for any control or storage across messages.

$$\underline{\text{LINE}} \quad (\underline{\text{TALLY}} \quad [\left. \begin{array}{c} 0 \\ 1 \end{array} \right\}])$$

These tallies may be referenced in both line control and request procedures in ASSIGNMENT and logical statements. In two-way simultaneous procedures, these tallies are implicitly the primary line's tallies.

Examples

```
LINE (TALLY[1]) = RETRY - 1.  
IF LINE (TALLY[1]) = 5 THEN GO TO 12.
```

AUXILIARY LINE REGISTERS

In full-duplex procedures, a second set of elements is emitted by the NDL compiler (c.f., Line section, TYPE statement) to provide for the management of the host input side or auxiliary line. As in the primary line table, these are general purpose byte-variables and may be used for whatever use invented by the NDL programmer. The auxiliary line is associated with the input side of a full-duplex line.

$$\left\{ \begin{array}{c} \underline{\text{AUX}} \\ \underline{\text{AUXILIARY}} \end{array} \right\} \quad (\text{LINE}(\text{TALLY} [\left. \begin{array}{c} 0 \\ 1 \end{array} \right\}]))$$

Examples

```
IF AUX(LINE(TALLY[0])) GTR 0 THEN  
AUXILIARY (LINE(TALLY[1])) = 1.
```

RETRY REGISTER

The contents of this register is initially set by the NDL programmer with the RETRY statement in the NDL Station section. The number of actual retries that were performed is available to the user. The value of retry is re-initialized upon execution of a TERMINATE ERROR, TERMINATE NORMAL, or TERMINATE BLOCK or execution of an INITIALIZE statement.

RETRY

The maximum value that may be specified for a retry value is 255 decimal. In line control it has a valid value only if the station processed at the time is valid. Each station table has a retry value stored in it. The

**RETRY REGISTER
STATION FREQUENCY REGISTER
STATION REGISTER
TOGS**

RETRY statement is generally used in a request procedure.

Example

```
IF RETRY GT ZERO THEN
  BEGIN
    RETRY = RETRY - 1
    TERMINATE NOINPUT.
```

STATION FREQUENCY REGISTER

This register is the read-only value corresponding to the value which is declared in the FREQUENCY statement in the Station section. It may be referenced in both a line control and request procedure.

STATION (FREQUENCY)

Examples

```
STATION(TALLY) = STATION(FREQUENCY).
STATION(TALLY) = STATION(TALLY) - 1.
```

The STATION (FREQUENCY) may be changed via the MPL statement REDEFINE.STATION.

STATION REGISTER

This register contains an implicit index to the stations on the communications subsystem. Station may not be referenced in a request procedure.

STATION

If the station index is assigned to a value out of range, that is, a value which is greater than or equal to MAXSTATIONS, the station status is not valid and not ready, and references to station table items are inhibited.

Examples

```
IF STATION = ZERO THEN GO TO 14.
STATION = STATION - 1. GO TO 01.
14: STATION = MAXSTATIONS.
```

TOGS

These provide a means of referencing the entire user tog array as a byte variable.

TOGS

Examples

```
IF TOGS = KON1 THEN TOGS = 4'00'.
TOGS = 'F'.
TOGS = 4'FF'.
TOGS = BCC
```

STATION TALLY REGISTER

There is one of these registers for each station on the system. It is a general purpose register and may be used by the NDL programmer for any desired control or storage across messages. This register is line table resident.

STATION (TALLY)

This tally may be referenced in both line control and request procedures in ASSIGNMENT or logical statements.

Examples

```
STATION(TALLY) = STATION(TALLY) + 1.  
IF STATION(TALLY) CTR 5 THEN GO TO 7.
```

USER TALLY REGISTERS

There are at least three of these registers associated with each station that provide the user with at least three separate bytes that may be used in any way. An additional sixteen tallies may be referenced only if the TALLIES statement is set true. (Refer to Station section, Section 7.) The maximum decimal value that may be stored in each of these tallies is 255. If a number that exceeds 255 is assigned to a tally, the left-most bits are truncated during the decimal-to-binary conversion and assignment process. User tally registers are station table resident.

```
TALLY [< integer > ]  
  < integer > = 0 to 18  
              inclusive
```

The bytes are referenced by a tally number corresponding to the separate bytes. The first three user tallies (0,1,2) may be changed by the MCS or NDL ASSIGNMENT statement, while the remaining sixteen user tallies (3-18) may only be changed by an NDL ASSIGNMENT statement. The first three user tallies (0,1,2) are returned with each result header for MCS information, while the remaining sixteen (3-18) are not returned with each result header. All user tallies are maintained across messages on a station-by-station basis. They may be referenced in both line control and request procedures. The first three tallies (0,1,2) must be stored (to be passed to the MCS) and initialized (to be made available to NDL.)

Note

The first three tallies (0,1,2) are located in every station table. The remaining 16 tallies (3-18 inclusive) are only located in station tables which have tallies set true. (Refer to TALLIES statement in Section 7.)

Examples

1. CHAR = TALLY[0].
2. STORE TALLY[2].
3. INITIALIZE TALLY[1].

Example 1 is a simple assignment causing the value user TALLY[0] to be copied into the CHAR register.

Example 2 causes the current value in user TALLY[2] of the station being addressed to be copied into a result header for return to MCS.

Example 3 copies the value in a message header defined for the purpose into the user TALLY [1] of the station being addressed or if no header is present, and sets the user TALLY[1] of the station currently addressed to zeros.

MAXSTATIONS REGISTER
SKIPCONTROL
IR

Section 2
NDL Language Elements

MAXSTATIONS REGISTER

The maxstations register is the maximum number of stations to be associated with the line, as declared for that line in the Line section. Maxstations is a read-only register and may be referenced in both request and line control procedures. Maxstations register is line table resident.

MAXSTATIONS

Examples

```
STATION = STATION + 1.  
IF STATION NEQ MAXSTATIONS  
THEN GO TO 11.
```

SKIPCONTROL

This is an eight-bit read only register maintained by the MCS via the message header.

SKIPCONTROL

The value of skipcontrol is used in conjunction with the skip and space toggles (refer to variant toggles).

IR

This register is a hardware register that provides the NDL programmer with access to various system status flags.

IR [bit-index]

The bits in IR are used as follows:

BIT	MEANING
0	RCV-DATA-LINE is in a space condition.
1	Change in line status has occurred since this bit was last tested (mark to space or space to mark).
2	Dataset is ready.
3	Data-Carrier detect is true.
4	Downstream request-to-send is true.
5	Ring-indicator is true.
6	Clear-to-send is true.
7	Unused.

IIR

This register is supplement to IR and is used in the same manner.

IIR [bit-index]

The bits in IIR are used as follows:

BIT	MEANING
0	Data-line-occupied (ACU) is true.
1 to 7	Reserved.

LCHAR

This register contains the image of the latest received character in its transmitted form.

LCHAR

The value of this register depicts the character before any translation is done.

TOGGLES

These are single bit fields that can assume a value of true or false only. All toggles are tested independently. Toggles are divided into different function areas.

toggle = {
error-flag-toggle
binary-toggle
crc-toggle
line-toggles
line-status-toggles
NORESPONSE
NOSPACE
request-toggles
station-status-toggles
syncs-toggle
underflow toggle
user-toggles
variant-toggles

Byte-Variable Bit Reference

Each of the eight (0-7) bits of a byte-variable may be interrogated and, if the particular byte-variable can be assigned a value, so may each bit be set true/false. The syntactic device to address the bits consists in suffixing a bracketed bit number to the byte reference. The right-most low order bit is 0.

Examples

```
STATION(TALLY)[1] = TRUE.  
AUX(LINE(TALLY[0]))[7] = FALSE.  
IF MAXSTATIONS[0] THEN  
IF CRC[1] [2] THEN CRC[0] [4] = FALSE.
```

Toggles may be reference toggles or assignable toggles. Assignable toggles may have their value changed by the NDL programmer. Reference toggles may only appear on the right side of the ASSIGNMENT by use

of the statement. Both assignable and reference toggles may be tested in an IF statement.

Assignable Toggles

TOG [tog index]
LINE (line-status)
CRC
SYNCS
Request-toggles
AUX (Line (line-status))
NOSPACE

Reference Only Toggles

Variant toggles
STATION (station-status)
UNDERFLOW

CRC TOGGLE

The CRC toggle provides the ability to exclude input or output characters from the CRC value accumulation. CRC block accumulation is declared in the PARITY statement (Terminal section), invoked by the use of the INITIALIZE statement (Line Control and Request section), altered by the SUM statement directly, and enabled/disabled by this toggle.

CRC

Examples

```
CRC = FALSE.  
TRANSMIT SOH.  
TRANSMIT ADDRESS (TRANSMIT).  
TRANSMIT STX.  
INITIALIZE CRC.  
CRC = TRUE.  
INITIALIZE TEXT.  
TRANSMIT TEXT.  
CRC = FALSE.  
TRANSMIT ETX.  
TRANSMIT CRC.  
FINISH TRANSMIT.
```

Note

When "CRC" appears as the operand of an INITIALIZE, TRANSMIT, or RECEIVE statement, it implicitly references the CRC register. When CRC appears as the left part of an assignment (CRC=TRUE), it is an implicit reference to the CRC toggle. The CRC register can only appear in an ASSIGNMENT statement left part as CRC [1] and CRC [0], referencing the high-order and low-order bits, respectively.

ERROR FLAGS

These references are not allowed in a line control definition. Error flags are divided into the following categories. (Bits designated in the header are associated with each error-flag toggle.)

error-flags = { item-error-flags
 receive-error-flags
 break-error-flags }

These error-flag-toggles are normally set by firmware upon detection of the condition. These error flags are reset by an ASSIGNMENT statement that references them or by any TERMINATE statement.

Item Error Flags

These toggles may not be used in control procedures except in RECEIVE statements.

BCCERR

This is true if the block check character (BCC) computed by the communications subsystem is not equivalent to the BCC received from a station. The parity of the BCC is defined in the PARITY statement of the terminal description associated with a station.

ADDERR

This is true on a RECEIVE ADDRESS statement when an error in the station address is detected. The size and value of the station address, as defined in the terminal description and station attribute list, allow the communications subsystem to check the correct value against the value received.

TRANERR

This is true on input from a station if the transmission number received is not one greater than the number of the previously received message from the same station.

ENDOFBUFFER

In the case of a receive request, this occurs when the RECEIVE TEXT exceeds the maximum allowed. The maximum allowed size is specified by the MAXINPUT statement in the terminal description.

CRCERR

If true (is set implicitly), this indicates that the CRC received was not equal to the CRC that was locally computed.

During bit-oriented communications, if true, indicates an end flag is detected and the frame is rejected. However, if during a receive CRC, the toggle is not set, the CRC is valid, but more than two characters are received.

FORMATTER

This is true if an NDL request used the RECEIVE STRING option to input data to the communications subsystem and the characters received do not correspond to the string specified.

Receive Error Flags

These toggles may not be used in control procedures except in RECEIVE statements and ERROR SWITCH statements.

PARITY

As its name implies, PARITY is true when a parity error is detected by the communications subsystem while receiving a message, but only if PARITY has been defined by a PARITY statement in the terminal description for the station.

STOPBIT

This is true if the communications subsystem is expecting to receive a STOPBIT on an asynchronous line as the next input on that line and, due to parity or noise, the STOPBIT is not received. This toggle may not be specified if the bits toggle is specified.

TIMEOUT

This is true if a character is not received from a station within the time specified in the TIMEOUT statement of the terminal description for a station; i.e., maximum time between input characters or in the receive statement being executed.

BREAK

This is true if a BREAK was received by the communications subsystem.

BUFOVFL

This is true if the communications subsystem was unable to serve the interrupt before the next character was received on the line, thereby having destroyed the previous character.

LOSSOFCARRIER

This bit is set true if loss of carrier LOSSOFCARRIER is detected by the firmware.

ABORT

This is true if the communications subsystem receives 7 to 14 continuous one-bits. This toggle can only be referenced in procedures used by a terminal of type bits and takes the place of STOPBIT. (Refer to TYPE statement in Section 6.)

IDLE

This is true if the communication subsystem detects 14 contiguous one-bits while in receive. This toggle can only be referenced in procedures used by a terminal of type bits and take the place of BREAK-ON-RECEIVE. (Refer to TYPE statement in Section 6.)

INVALID

This is set true if the communications subsystem detects the following conditions:

1. An end flag is sensed during a receive other than RECEIVE TEXT or RECEIVE CRC.
2. An end flag is sensed during a RECEIVE TEXT that checks for a text control character.
3. An end flag is sensed before two characters are received or an invalid CRC is received during a RECEIVE TEXT which does not check for text control character.
4. During a RECEIVE CRC and the CRC is valid but more than two characters are received.

This toggle can only be referenced in procedures used by a terminal of type bits and takes the place of PARITY. (Refer to TYPE statement in Section 6.)

Break Error Flags

BREAK[RECEIVE]

Refers to BREAK ON INPUT. It may not be used in control procedure.

BREAK[TRANSMIT]

Refers to BREAK ON OUTPUT. It may not be used in control procedures.

LINE TOGGLES

In the line table that is created for each line by the NDL compiler, two bits have been reserved as line toggles.

$$\text{line-toggle} = \underline{\text{LINE}} \quad \left(\begin{array}{c} \underline{\text{TOG}}[0] \\ \underline{\text{TOG}}[1] \end{array} \right)$$

These two line toggles are general purpose bit variables. These may be used for any purpose by the NDL programmer. Line toggles reset by the "MAKE LINE READY" message from the MCS, if the line was in "not ready" state.

AUXILIARY LINE TOGGLES

If in the Line section, a duplex declaration is specified, an Auxiliary Line Table exists and AUXILIARY (LINE(TOG[0 or 1])) is liable to interrogation and alteration.

$$\text{Aux-line-toggle} = \underline{\text{AUXILIARY}} (\underline{\text{LINE}} (\left\{ \begin{array}{l} \underline{\text{TOG[0]}} \\ \underline{\text{TOG[1]}} \end{array} \right\}))$$

Examples

```
IF LINE (TOG[0]) THEN
  AUX(LINE(TOG[1])) = TRUE.
AUXILIARY(LINE(TOG[0])) = FALSE.
```

LINE STATUS TOGGLES

Each line has two status toggles associated with it. These toggles are maintained in its associated line table.

$$\text{line-status-toggles} = \underline{\text{LINE}} (\left\{ \begin{array}{l} \underline{\text{BUSY}} \\ \underline{\text{QUEUED}} \end{array} \right\})$$

BUSY

The LINE (BUSY) status toggle allows system contention for the line; that is, the processing of any MCS queued functions for that line. If a line is not busy, the MCS may initiate and perform functions. If a line is busy, the MCS functions are queued until the line is available (not busy). The LINE (BUSY) status toggle can be set or reset explicitly by the NDL programmer. It is set implicitly by an MPL QUEUE statement, NDL FORK statement, or when line control is initiated and is implicitly reset by a NDL terminate instruction.

AUX(LINE)STATUS TOGGLES

In the case of two-way simultaneous procedures, if LINE (BUSY) is false, a FORK statement executed by the AUX line causes the primary LINE(BUSY) to be set. If AUX(LINE(BUSY)) is false and the primary executes a FORK statement, AUX(LINE(BUSY)) is set.

When LINE (BUSY) is set true, interruption of a primary control or primary request procedure by the auxiliary is inhibited.

The AUXILIARY line is associated with the input side of the line.

QUEUED

The LINE QUEUED status toggle is set true (implicitly) if a message is queued for a ready station on a ready line, or when a station on a ready line is made ready or enabled. It is the NDL programmer's responsibility to reset LINE (QUEUED). AUX(LINE (QUEUED)) is totally under NDL programmer control.

$$\text{aux-line-status toggles} = \underline{\text{AUXILIARY}}(\underline{\text{LINE}} (\left\{ \begin{array}{l} \underline{\text{BUSY}} \\ \underline{\text{QUEUED}} \end{array} \right\}))$$

AUX(LINE)STATUS TOGGLES
NORESPONSE
NOSPACE
REQUEST

Examples

LINE (QUEUED) = FALSE .
IF NOT AUX(LINE(BUSY)) THEN FORK 22.

NORESPONSE

This line-oriented toggle indicates that a timeout interrupt has occurred from the gross timer. This toggle is used primarily by bit-oriented line disciplines. Although the toggle is set by the communications subsystem, it can be reset by the load timer instruction. The NORESPONSE toggle may be accessed at any time by either co-line of a full-duplex pair.

NORESPONSE

NOSPACE

This toggle is used to determine if space is available for a particular station. When a check for NOSPACE is made, the system checks that enough buffer space is in the available buffer poll and that the MCS input queue limit for the station has not been exceeded.

NOSPACE

If this toggle is set by a lack of available buffers, it is reset by available buffers. A NOSPACE indication caused by exceeding the MCS input queue limit for a particular station is reset by the MCS construct CONTINUE.STATION. CONTINUE.TASK resets NOSPACE caused by exceeding the MCS input queue limit for a particular task.

REQUEST

These toggles may not be referenced in a Line Control section. All of the request toggles may be set, reset, and tested independently. Request toggles are used in controlling the line.

request-toggle = $\left\{ \begin{array}{l} \underline{\text{NAKONSELECT}} \\ \underline{\text{NAKFLAG}} \\ \underline{\text{CONTROLFLAG}} \\ \underline{\text{WRUFLAG}} \\ \underline{\text{BLOCK/BLOCKED}} \\ \underline{\text{TRANSPARENT}} \\ \underline{\text{EVENT1}} \end{array} \right\}$

NAKONSELECT

This denotes that NAK (to a select message) was received and is maintained explicitly by the NDL programmer.

NAKFLAG

This toggle is maintained by the NDL programmer. It may be used to indicate that a transmitted output was NAK-ed.

CONTROLFLAG

This toggle is set true (implicitly) if, in the RECEIVE statement, the receive action specifies a check and receive for the control character that was defined for that station.

WRUFLAG

This toggle is set true (implicitly) if, in the RECEIVE statement, the receive action specifies a check and receive for the WRU (who-are-you) character that was defined for that station.

BLOCK/BLOCKED

This toggle indicates that this message is only a part of a larger logical record. It is implicitly set on input when a TERMINATE BLOCK statement is executed.

TRANSPARENT

This toggle indicates that this message is not to be translated on output or was not translated on input. This toggle is set by the NDL programmer.

EVENT1

This is a general-purpose toggle available to the NDL programmer.

STATION STATUS TOGGLES

These toggles are read-only to the NDL programmer; that is, they may be used only in logical expressions (IF statements) or only to the right of the equal (=) sign in an ASSIGNMNET statement.

$$\text{station-status-toggle} = \text{STATION} \left(\left\{ \begin{array}{l} \text{READY} \\ \text{QUEUED} \\ \text{ENABLED} \\ \text{VALID} \end{array} \right\} \right)$$

READY

The communications subsystem never attempts to automatically initiate any function for stations that are not ready. With the definition of a valid station status, the communications subsystem may distinguish between invalid stations and valid ones which are not ready. Although the communications subsystem does not automatically initiate any functions for not ready stations, the communications subsystem is guaranteed a station table to read or update for valid stations, ready or not ready. If a station is marked invalid, it is also marked not ready.

The ready station status causes the same action to occur as for the make station ready/not ready type (MCS write) or TERMINATE ERROR statement. The ready station is a necessary condition for initiating REQUESTS or ENABLEINPUT.

The station (ready) toggle is implicitly set false if:

1. TERMINATE ERROR statement is executed.
2. TERMINATE NOLABEL statement is executed.

The ready status toggle may be reset or set by means of a user command.

QUEUED

This toggle is true if the station queue contains at least one user function that could not be handled by the communications subsystem at the time the function was issued (LINE BUSY).

ENABLED

This toggle is true/false as dictated by the MCS QUEUE statement (enable input/disable input). This toggle must be set true before an INITIATE ENABLEINPUT can be executed.

VALID

This toggle is used to indicate that the station variable index is pointing to a valid station for that line for initiating request for that station.

Example

```
CONTROL TOP2BOTTOM:

    LINE(QUEUED) = FALSE.
    IE LINE (TOG[1]) THEN GO TO 20.
%   IF THE STATION HAS MORE THAN 1 MSG TO SEND TO
%   THE HOST. THE REQUEST WILL SIGNAL
%   THIS LINE CONTROL WITH A NON-ZERO STATION TALLY.
    IF STATION (TALLY) GTR 0 THEN GO TO 40.
5:   STATION = MAXSTATIONS - 1.
10:  IF NOT STATION (VALID)
    THEN GO TO 100.
20:  IF NOT STATION (READY)
    THEN GO TO 100.
30:  IF STATION (QUEUED) THEN
    BEGIN
        LINE (TOG[1]) = TRUE.
        INITIATE REQUEST.
        GO TO 20.
    END.
    LINE (TOG[1]) = FALSE.
40:  IF STATION (ENABLED) THEN
    INITIATE ENABLEINPUT.
100: IF STATION GTR 0 THEN
    BEGIN
        STATION = STATION - 1.
        GO TO 10.
    END.
    IF LINE (QUEUED) THEN GO TO 5.
    IDLE.
```

SYNCS

When this toggle is true, recognition of sync characters is inhibited so sync characters are passed through the RECEIVE statement as part of the text. If syncs is false, all sync characters are automatically removed. Syncs is an assignable toggle.

SYNCS

Example

```
SYNCS = FALSE.
```

USER

In the header of each message, eight bits have been reserved as user toggles. These toggles may be used by the message control system and, in NDL requests, as supplementary flags. User toggle settings may be carried between inputs as they are maintained on a station basis. The format for user toggle representation is as follows.

$$\text{user-toggle} = \underline{\text{TOG}} \left[\begin{array}{c} 7 \\ 6 \\ 5 \\ 4 \\ 3 \\ 2 \\ 1 \\ 0 \end{array} \right]$$

Example

TOG[5]

Note

TOG [integer] is a reference to that tog value in the currently addressed station table. Values are assigned to togs via INITIALIZE and ASSIGNMENT statements. TOG [7] is the left-most or high-order bit.

UNDERFLOW

This toggle is a reference-only toggle that is associated with each station. It is set true during synchronous operation whenever a character is transmitted too late.

UNDERFLO

VARIANT

These toggles are part of the message header prefixing an output request. They are set by MCS and may only be interrogated by an NDL request set.

Variant toggles, when true, indicate if certain device dependent operations are to be performed. References to variant toggles in line control are not allowed.

$$\text{variant-toggles} = \left\{ \begin{array}{c} \underline{\text{SKIP}} \\ \underline{\text{PAGE}} \\ \underline{\text{CARRIAGE}} \\ \underline{\text{LINEFEED}} \\ \underline{\text{PAPERMOTION}} \\ \underline{\text{SPACE}} \end{array} \right\}$$

SKIP

When true, indicates that a value is present in the variant, read-only byte, SKIPCONTROL. The NDL request set is free to take whatever action is appropriate if SKIP is true.

PAGE

When true, indicates page format control is to be done.

CARRIAGE

When true, indicates that a carriage return format is to be sent. The true state of this bit is 0 or reset. When CARRIAGE appears in an IF statement, the compiler generates IF NOT CARRIAGE.

LINEFEED

When true, indicates that line feed format control information is to be sent. The true state of this bit is 0 or reset. When LINEFEED appears in an IF statement, the compiler generates IF NOT LINEFEED.

UNDERFLOW

PAPERMOTION

When true, indicates that a new line should be started before the next line is transmitted.

SPACE

When true, indicates that horizontal spacing of SKIPCONTROL number of characters can be done.

The above descriptions represent the conventional application of these toggles. Each name (SKIP, etc.) is converted by the compiler into a bit address of a variant toggle appearing in the output message header generated by an MCS. Any communication use, from MCS to NDL, capable of expression in eight toggles and one byte, can be served.

SECTION 3

CONSTANT AND TRANSLATION SECTION

GENERAL

This section provides a method for equating user-defined identifiers with data strings. The occurrence of the user-defined identifier outside of the Constant section is equivalent to the occurrence of the string defined by that identifier.

CONSTANT SECTION DESCRIPTION

The format for this section is:

<u>CONSTANT</u>	{	identifier-1	=	"character"
	}	identifier-2	=	4 "hex-string".
		identifier-n	=	" ".

A simple hexadecimal string should consist of an even number of hexadecimal characters; odd-length strings are zero-padded at the left. When transmitted, data strings are translated to the appropriate code required by the remote device.

The maximum length of a string is 248 characters or 248 hexadecimal elements. Hexadecimal character strings must have a 4 preceding the first quote marks.

Examples

```
CONSTANT
  NUL = 4'00', % HEX STRING
  STX = 4'02',
  ERR1 = "ERROR1", %
  ERR2 = "ERROR2":
```

Constant definitions are separated by commas and more than one definition may be placed on a card. The Constant section is terminated by a period (.) following the last defined identifier.

Note

When a hexadecimal representation of a character is desired, the ASCII representation for that character must be used (i.e., POL = 4'70').

TRANSLATION SECTION DESCRIPTION

General

This section provides the NDL programmer with the ability to specify non-standard translation tables to transform characters to and from internal machine representation. The format of this section is:

TRANSLATION table-id: BYTE=integer

{ DEFAULTCHAR = variables.
SHIFT = variables.
empty. }

TRANSLATE = variables.

The Translation Table consists of two or more translate modes. Two most common modes are the input and output modes. Other modes are implemented whenever shift modes are included in the translation. There are four shift positions permitted for a character set: up, down, middle, and allcase. Allcase is when the character is common to all cases; therefore, no shift is ever necessary on output. Thus, there are five possible translation modes:

1. Upper → Internal.
2. Lower → Internal.
3. Middle → Internal.
4. Allcase → Internal.
5. Internal → Output.

Shift-mode translation is required when a subset of data is liable to different interpretation. In order to detect which shift mode translation applies at any particular time during data transfers, a shift mode flag is a typical convention used to precede the data, such as FIGS character in Baudot code.

If there are no shift modes in the character set, only two modes are possible:

1. Input → Output.
2. Output → Input.

The use of a translation table is invoked by the CODE statement in the Terminal section. The table-id specifies the name associated with each user defined translation table.

Note

DEFAULTCHAR and SHIFT, if used, appear prior to any translation variable.

TRANSLATION BYTE

This statement specifies the number of bits in a character. Parity bits are not included in this value. The format is:

BYTE = integer

This statement must appear prior to a TRANSLATE statement. The maximum size for a character is eight bits, except when a multibase character set is being defined, in which case the maximum size is six bits. This statement is a required statement.

Example

BYTE = 7.

TRANSLATION SHIFT

The SHIFT statement specifies the characters that are to be used to cause the translation process to switch from one case to another. The format is:

$$\underline{\text{SHIFT}} = \left\{ \begin{array}{l} \underline{\text{UPPER:}} \\ \underline{\text{LOWER:}} \\ \underline{\text{MIDDLE:}} \end{array} \right\} \text{unitary-string} \left. \vphantom{\begin{array}{l} \underline{\text{UPPER:}} \\ \underline{\text{LOWER:}} \\ \underline{\text{MIDDLE:}} \end{array}} \right\} , \dots$$

This SHIFT statement must appear prior to a TRANSLATE statement and is required when a multibase character set is being defined.

Example

SHIFT = UPPER : 4'06'', LOWER : 4'09''.

Note

Refer to the description of SHIFT statement in Section 4.

TRANSLATION DEFAULTCHAR

The DEFAULTCHAR statement specifies to which character all translation table entries are initialized. Any translations not explicitly specified in a TRANSLATE statement translates into the character declared in this statement. This format is:

$$\underline{\text{DEFAULTCHAR}} = \left\{ \begin{array}{ll} \underline{\text{INPUT:}} & \text{unitary string} \\ & \underline{\text{(UPPER)}} \\ & \underline{\text{(LOWER)}} \\ \underline{\text{OUTPUT:}} & \underline{\text{(MIDDLE)}} \text{unitary-string} \\ & \underline{\text{(ALLCASE)}} \\ & \text{empty} \end{array} \right\} \dots$$

The input clause specifies to which undefined incoming characters are translated. The output clause specifies to which undefined outgoing characters are translated. The default input character is a question mark (?), and the default output character is a NULL (4'00''). For a multi-case character set, translate mode specifies the case of the output default character.

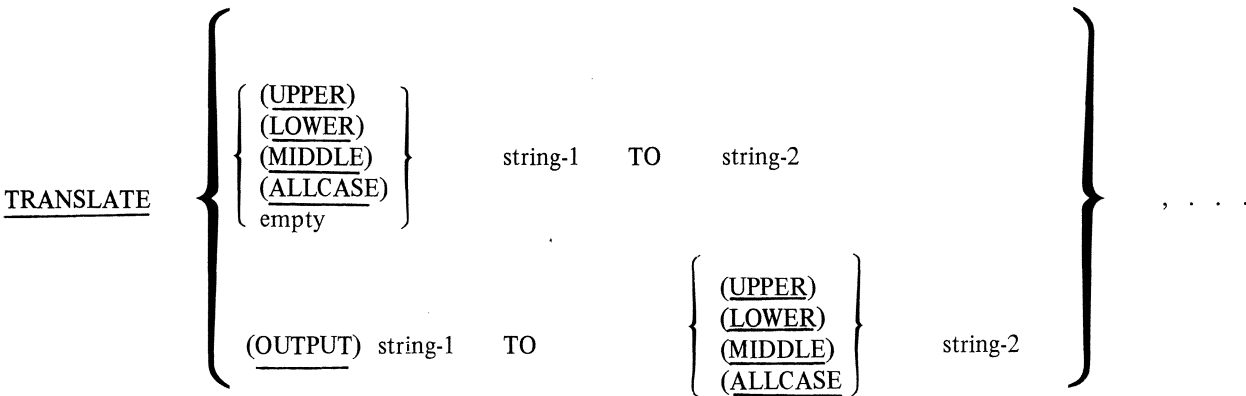
The DEFAULTCHAR statement must appear prior to a TRANSLATE statement.

Examples

DEFAULTCHAR = 4"3F".
 DEFAULTCHAR = INPUT: "?".
 DEFAULTCHAR = OUTPUT: (MIDDLE) 4"37", INPUT: "?".

TRANSLATION TRANSLATE

This statement specifies the actual translations to be performed on input and output. In a single case character set, translate mode must be empty. Each "string TO string" construct specifies that each received character (which appears in the first string) is translated into the corresponding character in the second string. If a received character does not appear in the TRANSLATE statement, it is translated to the DEFAULTCHAR statement.



The output translation is calculated from the input specification. However, if two or more input characters translate into the same character, the output form of the translate element is used to specify the proper output character translation. This form permits resolution of conflicting input translations and allows non-symmetrical translation on output (i.e., on input "A" translates to "B" and on output "B" translates to "C", instead of "A").

When defining a multi-case character set, translate mode specifies in which case the character belongs. If translate mode is empty, it defaults to the most recently specified translate mode. If translate mode is empty for the first string, ALLCASE is assumed. ALLCASE specifies that a character is common to each case. The ALLCASE default also applies to the first output clause without a specified case.

Each string in a pair must be the same length in bytes. An odd length hexadecimal string has a zero inserted in the first digit position.

The TRANSLATE statement must appear last in a translation table definition; it is a required statement.

Example

```
TRANSLATE
  (ALLCASE) 4"000204081B1F20242E32333B3F" TO
            4"000D200A00000711001205087F",
            4"101112131415161718191A1C1D" TO
            4"000D200A00000711001205087F",
  (LOWER)   4"0103050607090A0B0C0D0E0F" TO
            4"3F3F62676C3F683F63673F6D",
  (LOWER)   4"21222325262728292A2B2C2D2F" TO
            4"693F773F3F617A3F723F3F7964",
            4"30313435363738393A3C3D3E" TO
            4"3F3F3F6E3F7678733F3F3F3F",
  (UPPER)   4"0103050607090A0B0C0D0E0F" TO
            4"524947424E4B4546554F5843",
            4"101112131415161718191A1C1D1E" TO
            4"523F53564154445A3F5148593F4A",
  (UPPER)   4"21222325262728292A2B2C2D2F" TO
            4"3426383F4D3F503F333F37394C",
            4"30313435363738393A3C3D3E" TO
            4"323F3F353F3F30313F363F3F",
  (MIDDLE)  4"0103050607090A0B0C0D0E0F" TO
            4"343862293E2E333F3739282C",
            4"101112131415161718191A1C1D1E" TO
            4"322F3F652D353F2B2C3171363F3A",
  (MIDDLE)  4"21222325262728292A2B2C2D" TO
            4"233D3F3F2A303F253F402464",
            4"30313435363738393A3C3D3E" TO
            4"3F3F3F3F3F3F5F033F3F3F3F",
  (OUTPUT)  4"00" TO (ALLCASE) 4"00",
            4"2C" TO (MIDDLE) 4"0F",
            4"3F" TO (MIDDLE) 4"37",
            4"62" TO (LOWER) 4"05",
            4"64" TO (LOWER) 4"2F",
            4"65" TO (LOWER) 4"13",
  (OUTPUT)  4"30313233343536373839" TO
            4"3839302A21353C2C232D".
```


ACU

The ACU statement allows the NDL programmer to condition the hardware by loading a descriptor to the line adapter.

ACU = Value

Each bit of the value raises or lowers the corresponding ACU signal as shown below:

Bit	Meaning
0 (right-most)	Call request (CRC)
1 to 7	Reserved

ADDRESS

address-function = RECEIVE { (time) } { empty }

ADDRESS { (TRANSMIT) } { (RECEIVE) } { empty }

{ [{ ERROR[error-id] } { error-id } { empty }] , ADDERR : { NULL } { label } } { empty }

This is a special case of NDL syntax, allowed in byte-variable assignment and IF statements in a line control set only. (Not allowed in IF statements or byte-variable assignments in request sets.) Functions are to:

1. Accept the specified ADDRESS from a station;
2. Scan all station tables resident on that line and validate the RECEIVED ADDRESS; and
3. Either: (a) assign the station table index to the byte-variable if the ADDRESS is found in a station table; or (b) execute the ERROR action if specified, or mark the line not ready and return an ERROR result.

Note

Refer to the description of the RECEIVE statement.

The value of the character register is destroyed.

ASSIGNMENT

The ASSIGNMENT statement enables an NDL request set to assign and/or update a byte-variable or to condition a toggle. A toggle can only assume a value of true or false.

There are three formats for the ASSIGNMENT statement. The first is to make a toggle assume a value of true or false, or make one toggle assume the value of another toggle.

assignable-toggle = { toggle } { TRUE } { FALSE }

BINARY (REQUEST)

This statement provides the NDL programmer with the ability to inhibit the automatic character translation performed when characters are received or transmitted.

The format of the BINARY statement is:

$$\underline{\text{BINARY}} = \left\{ \begin{array}{l} \underline{\text{TRUE}} \\ \underline{\text{FALSE}} \end{array} \right\}$$

The FALSE option specifies that characters are to undergo translation to or from internal machine representation. The TRUE option inhibits any translation and parity checking.

Example

BINARY = TRUE.

CONTINUE

This statement is used in conjunction with the WAIT statement and the CONTINUE option of the RECEIVE statement. It enables code execution synchronization in control and request sets peculiar to two-way simultaneous (TWS) or duplex communication channels. In TWS channels, there are two active paths (or lines); CONTINUE enables an active side to prompt a WAITing side back into action.

CONTINUE

Example

IF NOT LINE (TOG[1]) THEN CONTINUE.

Note

If the "other" side is not WAITing, CONTINUE is a NO-OP.

DATASET

This statement allows the NDL programmer to condition the hardware by loading a dataset descriptor to the line adapter.

DATASET = Value

Each bit of the value raises or lowers a specified dataset signal as follows:

BIT	MEANING
0 (right-most)	Request-to-send (RTS)
1	Originate (ORIG)
2	Data-terminal-ready (DTR)
3	Data-mode (DM)
4	New-synchronous (NEW-SYNCH)
5 to 7	Reserved

DELAY

This statement specifies the time lapse before executing the next statement of the request or control sequence. It suspends activity on the line and yields control to the firmware to address interrupts and other processing management until the specified time expires when control is returned to the next sequential instruction.

$$\underline{\text{DELAY}} \quad (\quad \text{integer} \quad \left\{ \begin{array}{l} \underline{\text{MICRO}} \\ \underline{\text{MILLI}} \\ \underline{\text{SEC}} \\ \underline{\text{MIN}} \end{array} \right\})$$

The maximum allowable value for the DELAY statement is 65535 milliseconds.

Examples

DELAY (50 MILLI).
DELAY (1 MIN).
73. GETSPACE[94].
 INITIATE RECEIVE.

94. DELAY (200 MILLI).
 GO 73.

DELIVER

This statement is used exclusively by bit-oriented line disciplines. It gives the NDL programmer the ability to remove messages from the input save queue.

$$\underline{\text{DELIVER}} \left\{ \begin{array}{l} (\text{ < value > }) \left\{ \begin{array}{l} \underline{\text{NULL}} \\ \text{label} \end{array} \right\} \\ \text{ALL (DISCARD)} \end{array} \right\}$$

When using the DELIVER statement, if <value> is specified, that number of messages is removed from the top of the input save queue and queued to the bottom of the result queue. The input queue count is decremented by the <value>, which is a literal, or is derived from the contents of a specified byte variable. If <value> is greater than the queue count, a branch to the specified label is executed, and no messages are removed from the input save queue.

If <value> is zero, the instruction is a NO-OP.

If ALL is specified, then all messages are removed from the input save queue and queued to the result queue. The input save queue count is zeroed. If the input save queue count is zero, this instruction is a NO-OP.

If DISCARD is specified, all messages are removed but not queued to the result queue. The message space is released. If the input save queue count is zero, this instruction is a NO-OP.

The DELIVER statement is valid only in a procedure referenced by a terminal of type bits. (Refer to TYPE statement in Section 6.)

Examples

DELIVER (TALLY[0]) [1].
DELIVER (5).
DELIVER ALL (DISCARD).

DISABLE

This statement gives the NDL programmer the ability to control concatenated terminals.

The format for this statement is:

$$\text{DISABLE} \left\{ \begin{array}{l} \underline{\text{DOWNSTREAM}} \\ \underline{\text{TRANSMIT}} \end{array} \right\}$$

If DOWNSTREAM is specified, the request-to-send signal is blocked from passing upstream and inhibits the clear-to-send signal passing downstream on concatenated terminals.

If TRANSMIT is specified, then this statement causes the request-to-send signal to be dropped; this signal is blocked from passing upstream on concatenated terminals.

DISABLE, when executed by the auxiliary half of full-duplex pair, is a NO-OP.

Examples

DISABLE DOWNSTREAM.
DISABLE TRANSMIT.

DISCONNECT

This statement allows the NDL programmer to hang up a switched line (at one's own discretion) or make a leased line not ready. The line is marked not ready and the MCS is notified that the line has been disconnected.

DISCONNECT

ENABLE

This statement gives the NDL programmer the ability to control concatenated terminals.

The format of this statement is:

ENABLE { DOWNSTREAM (< time > , < label >) }
 { TRANSMIT empty }

If DOWNSTREAM is specified, the request-to-send and the clear-to-send signals pass upstream and downstream, respectively, from/to this terminal.

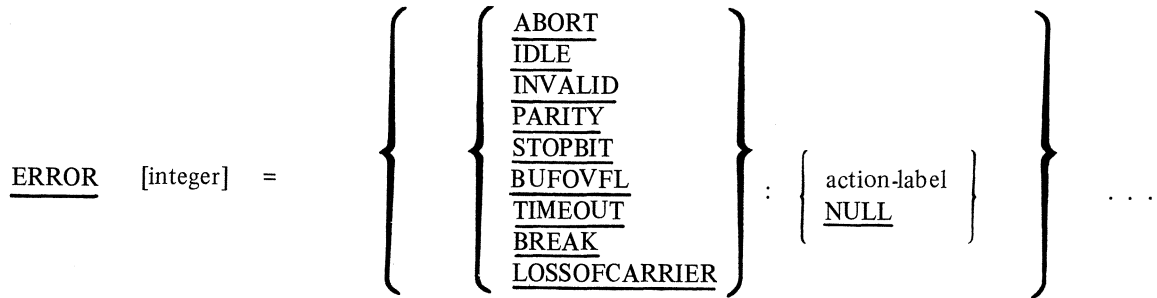
If TRANSMIT is specified, a request-to-send signal is raised. Procedure execution for this line is suspended until clear-to-send is true or the specified time expires. If clear-to-send becomes true, the next instruction in sequence is executed. If the timer expires, a branch to the label is taken. If empty, the next instruction in sequence is executed after raising request-to-send. When executed by the auxiliary half of a full-duplex pair, ENABLE is a NO-OP.

Examples

ENABLE DOWNSTREAM
ENABLE TRANSMIT (1 sec, 50).

ERROR

This statement provides: 1) a means to specify a table of receive type errors and associated transfer labels on error occurrence; and 2) a convenient shorthand reference to an error table within a RECEIVE statement. Only by specifying a transfer label can the NDL request set retain control to perform cleanup, pass signals to the MCS through user tallies and toggles, or retry the input. If a receive error occurs for which no transfer label was specified, the system marks the active station not ready, and returns a TERMINATE NOLABEL indication in the result header passed to the host.



Each control or request set can have a maximum of 40 error statements. This number includes implicit error switches declared in RECEIVE statements which directly specify all or part of the receive errors instead of referring to a declared error switch number.

If NULL is specified as an action label, the result is dependent upon the RECEIVE statement being executed. In the case of RECEIVE TEXT, the error is ignored and RECEIVE TEXT continues. In the case of any other type of receive, control is then transferred to the next instruction on error occurrence.

ABORT, IDLE, and INVALID can be specified only in a procedure that is used by a terminal of type bits. (Refer to TYPE statement in section 6). PARITY, STOPBIT, and BREAK cannot be used if the terminal is of type bits.

Examples

1. RECEIVETEXT [TIMEOUT:NULL,
LOSSOFCARRIER:102,
PARITY:103,
BUFOVFL:104].
2. RECEIVE [STOPBIT:NULL,
BREAK:123,
BUFOVFL:125,
LOSSOFCARRIER:173].
3. ERROR[2] = PARITY:10,
STOPBIT:11,
BUFOVFL:12,
TIMEOUT:13,
BREAK:14,
LOSSOFCARRIER:15.
- ERROR[1] = LOSSOFCARRIER:103,
PARITY:17,
BUFOVFL:12,
TIMEOUT:22.
- ERROR[4] = PARITY:39,
TIMEOUT:42,
LOSSOFCARRIER:103.
4. ERROR[1] = ABORT:720,
TIMEOUT:720,
BUFOVFL:700,
IDLE:605,
INVALID:720,
LOSSOFCARRIER:720.

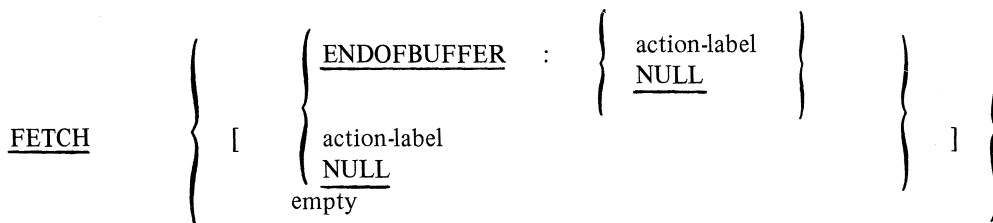
```
ERROR[3] = ABORT:50,
          TIMEOUT:60,
          BUFOVLF:70,
          IDLE:100,
          INVALID:50,
          LOSSOFCARRIER:55.
```

```
RECEIVE[ERROR[4]].
RECEIVE SOH[1].
RECEIVE TEXT[2].
```

FETCH

This statement loads the character register with the bytes in the message text buffer pointed to by the message pointer. FETCH is valid only for output from the communications subsystem to a station and may only be used in a Request section. (FETCH is a NO-OP in an input request.)

The format for the FETCH statement is:



When using the FETCH statement, provision should be made for taking action on an ENDOFBUFFER condition. (A FETCH is performed after the last character has been removed from the message text.) An ENDOFBUFFER condition on a FETCH statement is not considered an error. Thus, the ENDOFBUFFER flag is not set.

Successive FETCH statements cause the message pointer to be incremented by one.

Examples

```
FETCH [ENDOFBUFFER:3].
FETCH [3].
```

Both of these statements have the same result.

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

Both of these statements have the same result (a fall-through occurs).

```
FETCH.
```

In this example, no ENDOFBUFFER action is specified and a TERMINATE NOLABEL occurs.

Note

A TERMINATE NOLABEL results when an error occurs for which the programmer has no disposition. The MCS is informed through an error result.

FINISH TRANSMIT

This statement provides the delay time needed for the last character transmitted to propagate down the line before a LINE TURNAROUND is initiated. This command places the line back into receive mode.

The format for this statement is:

$$\underline{\text{FINISH TRANSMIT}} \left\{ \begin{array}{l} \text{(integer MICRO)} \\ \text{(integer MILLI)} \\ \text{(integer SEC)} \\ \text{(integer MIN)} \\ \text{(NULL)} \\ \text{empty} \end{array} \right\}$$

The FINISH TRANSMIT statement is used to get a line ready to receive (data or EOT) following a POLL. The time specified is the actual delay before turning the line around. If the time specified is NULL or 0 (zero), no delay occurs. If empty, a delay of two bit times at the specified line rate is used.

Examples

FINISH TRANSMIT.
FINISH TRANSMIT (8 MILLI).

FLAGFILL

This statement is used exclusively by bit-oriented line disciplines, and turns on automatic interframe time fill in the line adapter. A minimum of two flag sequences is transmitted. Flags continue to be transmitted until an INITIATE TRANSMIT, TRANSMIT, or FINISH TRANSMIT is executed.

The format for this statement is:

$$\underline{\text{FLAGFILL}} \left\{ \begin{array}{l} \underline{\text{[UNDERFLO}} : \left\{ \begin{array}{l} \text{NULL} \\ \text{Label} \end{array} \right\} | \end{array} \right\}$$

If a synchronous underflow is detected, a branch to the specified UNDERFLOW ADDRESS is performed.

FLAGFILL, when executed by the auxiliary half of a full-duplex pair, is a NO-OP.

The FLAGFILL statement can be executed only in a procedure reference by terminals of type bits. (Refer to TYPE statement in Section 6.)

Examples

FLAGFILL.
FLAGFILL [UNDERFLO : 50].

FORK

This statement is used in duplex communication control and request sets to engage the "other" line to begin code execution at the specified label. If the other line is busy, FORK is treated as a NO-OP. In either case, code execution continues at the next sequential instruction by the side which executed the FORK. Code execution by the "other" line begins if the other line is not busy, or as soon as the FORK-executing side fails into a yielding position, such as TRANSMIT, WAIT, or RECEIVE.

$$\underline{\text{FORK}} \left\{ \begin{array}{l} \text{label-id} \\ \text{empty} \end{array} \right\}$$

Either the primary or auxiliary code may execute a FORK. If the primary side executes a FORK, the intent is to direct the auxiliary to begin execution at label-id. FORK empty initiates the "other lines" line control procedure.

The Line section TYPE statement enables explicit declaration of the primary and implicit declaration of the auxiliary sides or lines of a TWS channel. The Terminal section CONTROL statement enables the explicit declaration of the primary control set and optional auxiliary control set and also enables naming the input and output request sets. The station declaration names an associated terminal declaration, and the Line section STATION statement names the stations resident on a line, thereby completing the circle of relationships.

Any MCS originated function intended to cause execution of a line control set exercising a duplex channel results in "waking up" the declared primary line control set. Therefore, the NDL programmer has two choices of levels at which to execute a FORK on the primary side code:

1. In the control set ("FORK label-id", where "label-id" points to a place in that control set which includes an INITIATE ENABLEINPUT statement).
2. In the output request set ("FORK label-id", where "label-id" points to a place in that request set which includes a RECEIVE statement).

All output (TRANSMIT statements) must be done on the primary line, and all input (RECEIVE statements) must be done on the auxiliary line. Any input or output to a station involves interaction which includes both TRANSMIT and RECEIVE. The FORK statement provides the ability, in an output request, to engage the AUX line, if not busy, and to execute the RECEIVE of a station message response (ACK, NAK, etc.). Conversely, FORK enables the input request set (being executed on the AUX line) to engage the primary to execute a TRANSMITTED message response to a station. Whenever it is necessary for one side of a duplex channel to perform an action not related to it (RECEIVE in the primary, TRANSMIT on the auxiliary), the NDL programmer can choose a signaling convention using the line and AUX line tallies and toggles, or directly engage the "other" side through the use of FORK.

Example

```

REQUEST DUPLXOUT:
=====
=====

FINISH TRANSMIT.
62. IF NOT AUX(LINE(BUSY)) THEN
    BEGIN
        FORK 101.
        WAIT.
    END
ELSE
    WAIT (500 MILLI).
    GO TO 62.
END.
=====
=====

101. INITIATE RECEIVE.
    RECEIVE[ACK:102. ERROR[2]].
=====
=====
    CONTINUE

102. TERMINATE.

```

In the previous example, the primary side of the line executes the statements from label 62 onwards. On detecting AUX line "not busy", the primary "forks" the auxiliary and then suspends itself. On execution of the WAIT statement by the primary, the auxiliary starts execution of code at statement 101. Execution of CONTINUE, by the auxiliary, causes the primary to restart at the statement following the WAIT. TERMINATE causes the auxiliary to go to the top of the auxiliary line control.

GETSPACE

This statement allocates buffer space in the processor main memory for the storage of input data. The action-label is taken if no buffer space is available. This statement is allowed only in an input request procedure.

The format for the GETSPACE statement is:

GETSPACE [action-label]

GETSPACE is a NO-OP if space already has been obtained; otherwise, it obtains space, if possible, or branches to the label specified, if not. In the POLL logic for a TC500, a GETSPACE statement is placed just before the RECEIVE TRAN statement, but is repeated again before the RECEIVE TEXT statement. Only enough space for one block of data is obtained.

Example

```
GETSPACE [2].  
RECEIVE TRAN [ERROR[0]. TRANERR:NULL].  
RECEIVE STX [ERROR[0]. FORMATERR:2].  
INITIALIZE TEXT.  
RECEIVE TEXT [ERROR[0]. ENDOFBUFFER:2].
```

In the above example, buffer space is allocated and the TRAN number is read and stored in the message header, and the text is stored in the space allocated.

Note

If a GETSPACE is not executed prior to a RECEIVE, the system attempts to get adequate space; if successful, the RECEIVE proceeds normally; if unsuccessful, a TERMINATE NO-LABEL occurs.

GO TO

This statement provides a way of breaking out of sequential, statement-by-statement execution and permits continuation at some other location indicated by the specified label.

The format for the GO TO statement is:

$$\underline{\text{GO}} \quad \text{TO} \quad \left\{ \begin{array}{l} \text{label} \\ \text{value: label list} \end{array} \right\}$$

The label may consist of up to five digits.

The first option is a simple GO TO. The second method is a variable GO TO. A list of labels is provided and the desired label is indicated by the value specified. A value greater than the number of labels is a NO-OP. A value of zero causes a branch to the first label in the list.

Examples

```
GO TO STATION(TALLY): 76, 91, 7.
```

Note

If STATION(TALLY) = 2, then a branch to label 7 is taken.

IDLE

This statement halts the processing of the line control procedure in which it is contained. IDLE causes the line to be marked not busy and left in a ready state. The IDLE statement is valid only in a line control procedure. This is the only way to do a terminate in line control.

IDLE

Upon execution of IDLE, the line is not busy and the next function to be initiated for that line automatically begins the line control at the first statement.

Example

```
IDLE.
```

IF**IF**

The function of this statement is to control the sequence of operations (as described by NDL statements) to be executed depending upon some relational conditions or true/false value.

The general format for the IF statement is:

$$\underline{\text{IF}} \text{ if-clause } \underline{\text{THEN}} \text{ statement} \quad \left\{ \begin{array}{l} \underline{\text{ELSE}} \text{ statement} \\ \text{empty} \end{array} \right\}$$

There are two variations of this format. The first option involves a test condition (true/false) of certain toggles and flags. The if-clause is replaced with the flag or toggle being tested.

The format for Option 1 of the IF statement is:

$$\underline{\text{IF}} \quad \left\{ \begin{array}{l} \underline{\text{NOT}} \\ \text{empty} \end{array} \right\} \left\{ \begin{array}{l} \text{toggle} \\ \underline{\text{request-toggles}} \\ \underline{\text{error-flags}} \\ \underline{\text{variant-toggle}} \end{array} \right\} \quad \underline{\text{THEN}} \text{ statement} \quad \left\{ \begin{array}{l} \underline{\text{ELSE}} \text{ statement} \\ \text{empty} \end{array} \right\}$$

The underlined lower case terms in the if-clause position may only be used in a Request section. The toggles and flags are explained in Section 2.

The second option pertains to a relational condition between two values.

The format for Option 2 of the IF statement is:

$$\underline{\text{IF}} \quad \left\{ \begin{array}{l} \underline{\text{NOT}} \\ \text{empty} \end{array} \right\} \quad \text{value relational-operator value}$$

$$\underline{\text{THEN}} \text{ statement} \quad \left\{ \begin{array}{l} \underline{\text{ELSE}} \\ \text{empty} \end{array} \right\} \text{ statement}$$

The relational version of the IF statement provides a way of comparing the relationship between the members of a set of values. A byte variable may be used as a value.

The possible relational-operators are:

GTR - Greater than.

GEQ - Greater than or equal to.

LSS - Less than.

LEQ - Less than or equal to.

NEQ - Not equal to.

EQL - Equal to.

> - Greater than.

< - Less than.

= - Equal to.

The following table lists the allowable entries in the IF statement and the section in which they are allowed. The entries in the table are explained in Section 2.

OPTION 1

	Entry	Request	Line Control
Toggle	TOG	X	X
	LINE [TOG(number)]	X	X
	NOSPACE	X	X
	STATION [TOG(number)]	X	X
	CRC	X	X
	SYNCS	X	X
Request-toggle	NAKONSELECT	X	
	NAKFLAG	X	
	CONTROLFLAG	X	
	WRUFLAG	X	
	BLOCK/BLOCKED	X	
	TRANSPARENT	X	
	EVENT1	X	
error-flags	PARITY	X	
	STOPBIT	X	
	BUFOVFL	X	
	TIMEOUT	X	
	BREAK	X	
	BITS	X	
	LOSSOFCARRIER	X	
	BCCERR	X	
	ADDERR	X	
	TRANERR	X	
	FORMATER	X	
	ENDOFBUFFER	X	
	BREAK RECEIVE	X	
	BREAK TRANSMIT	X	
CRCERR	X		
variant-toggle	PAPERMOTION	X	
	CARRIAGE	X	
	PAGE	X	
	LINEFEED	X	
	SKIP	X	
	SPACE	X	

OPTION 2

value	IR	X	X
	STATION (FREQUENCY)	X	X
	MAXSTATIONS	X	X
	CHARACTER	X	X
	STATION(TALLY)	X	X
	RECEIVE $\left\{ \begin{array}{l} \text{(Time)} \\ \text{(NULL)} \end{array} \right\}$ ADDRESS		X
	STATION		X
	RETRY	X	X
	BCC	X	X

**IF
INCREMENT
INITIALIZE**

	Entry	Request	Line Control	
value	{	LINE (TALLY[number])	X	X
		AUX (LINE(TALLY[number]))	X	X
		unitary-string	X	X
		integer	X	X
		TALLY(n)	X	X
		LCHAR	X	X
		SKIPCONTROL	X	X
		TOGS	X	X
		CRC[0]	X	X
		CRC[1]	X	X

Examples

Option 1:

```
IF TOG [6] THEN TRANSMIT LF.
IF BREAK THEN GO 86 ELSE
TRANSMIT EOT.
```

Option 2:

```
IF CHARACTER = NAK THEN GO 02.
IF RETRY GTR ZERO THEN GO 07
ELSE TERMINATE NOINPUT.
```

INCREMENT

This statement adds one (1) to the transmission number (TRAN).

The format for the INCREMENT statement is:

INCREMENT TRAN

No overflow is detected. If overflow occurs, the TRAN value becomes zero.

Example

```
INCREMENT TRAN
```

Note

TRAN INCREMENT IS MOD 10 (e.g. 012345678901234567890)

INITIALIZE

This statement causes the function specified to be initialized in the communications subsystem. The INITIALIZE statement has some restrictions in the Line Control section; that is, TEXT, TRAN, TALLY and TOG items may not be used.

<u>INITIALIZE</u>	{	<u>BCC</u> <u>CRC</u> <u>TEXT</u> <u>TRAN</u> <u>TOG</u> [integer] <u>TALLY</u> [integer] <u>RETRY</u> <u>TOGS</u>	}	...
-------------------	---	---	---	-----

INITIALIZE BCC causes the block check character for horizontal parity to be initialized in the communications subsystem. For even parity, this character is initialized to all zeroes; for odd parity, all ones. The BCC should always be re-initialized for error recovery.

The INITIATE RECEIVE form of the Request section INITIATE statement commands the initiation of receive delays as follows:

1. If the delay part is zero or null, receive delay is disabled and the line is enabled to the receive mode.
2. If the delay part is empty, the time-to-delay is the time stored in the station tables "initiate receive delay" value. This value is a result of the NOISEDELAY statement in the Modem section.
3. If the delay value is a specified time, then this is the delay time.

When condition 2 or 3 is present, the amount of time is the delay time prior to enabling the receive mode. This function is referred to as "sequelching" the line. Any data or other transitions on the line is discarded. If synchronous operations are present, this action initiates the required search for sync characters.

Time may be specified in microseconds, milliseconds, seconds, or minutes. The integer preceding the time delimiter is the number or time periods.

The INITIATE TRANSMIT statement sets the line adapter to a transmit state and implements delays. The delay part indicates the amount of time to delay prior to testing for the clear-to-send (CB) signal from the modem. After the delay is executed and CB is true, the next instruction is executed. If the delay part is NULL, no delay is used. If the delay part is empty, the time used is the value delay stored in the station tables ACTIVE TRANSMIT DELAY. This value is the greater of the terminal TURNAROUND time, or the line modem TRANSMIT delay or the station modem NOISEDELAY.

Examples

```
INITIATE REQUEST.  
INITIATE RECEIVE (10 MILLI).  
INITIATE TRANSMIT.
```

Note

For INITIATE RECEIVE (transmit to receive mode change) the default time value is the line modem NOISEDELAY value. For INITIATE TRANSMIT (receive to transmit mode change) the greater of the following is used as the default value:

1. Station Turnaround.
2. Station Noise Delay.
3. Transmit Delay.

LABEL

Statements in the Request and Line Control sections are executed sequentially as listed. When it is desirable to alter the order of execution of statements, the labeled statement is one way of doing this. The maximum length of a label is five digits. The label must be followed by a colon (:) to separate it from its associated statement. The GO TO statement is used to transfer control unconditionally to a labeled statement.

The format for using a label is as follows:

```
label: { statement  
        { compound-statement  
        { empty } . . .
```

Examples

```
86: BEGIN  
    IF STATION (QUEUED) THEN  
    GO TO 11.  
    END  
11: INITIATE REQUEST.
```

PAUSE

This statement temporarily suspends processing of NDL instructions to allow other line interrupts to be processed.

The format for the PAUSE statement is:

PAUSE

When a number of non-transmit or non-receive instructions are to be processed, a PAUSE must be inserted to provide processor time to the other lines in the system.

Control procedures that search a station list for something to initiate must include a PAUSE in the loop.

Example

PAUSE.

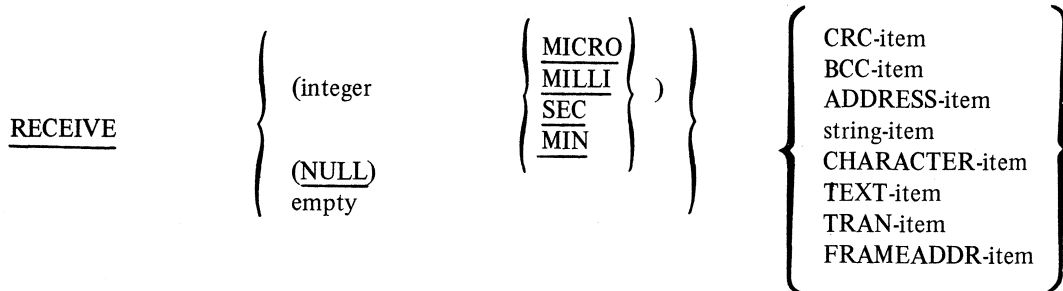
Note

Refer to the description of the DELAY statement.

RECEIVE

This statement accepts characters from a line and recognizes error conditions on a line. The line for which the RECEIVE statement is issued must be in a ready and receive mode.

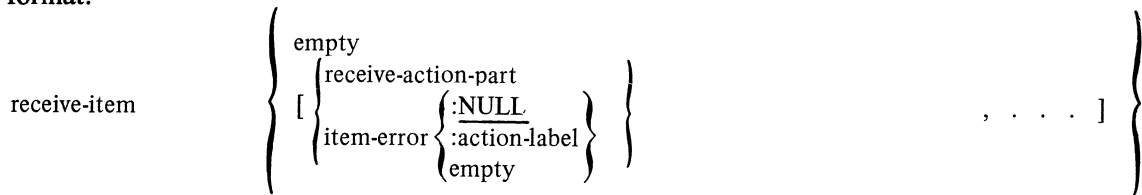
The general format for the RECEIVE statement is:



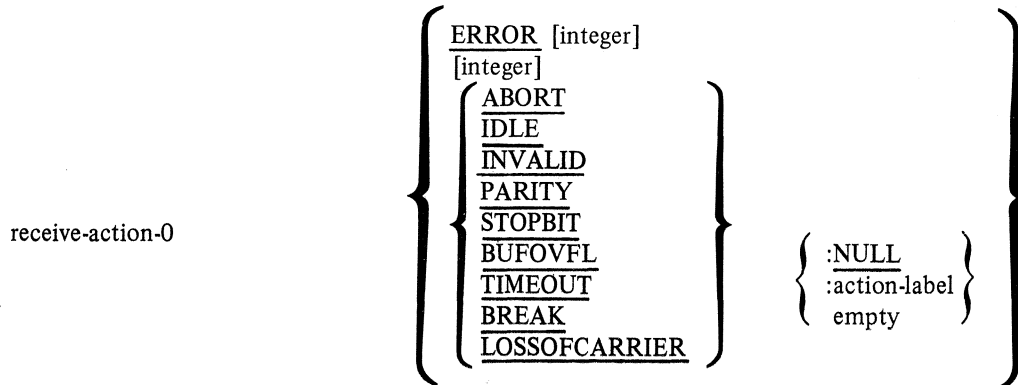
A RECEIVE statement receives the specified item. The "integer MILLI/MICRO/etc", if specified, replaces the terminal TIMEOUT value.

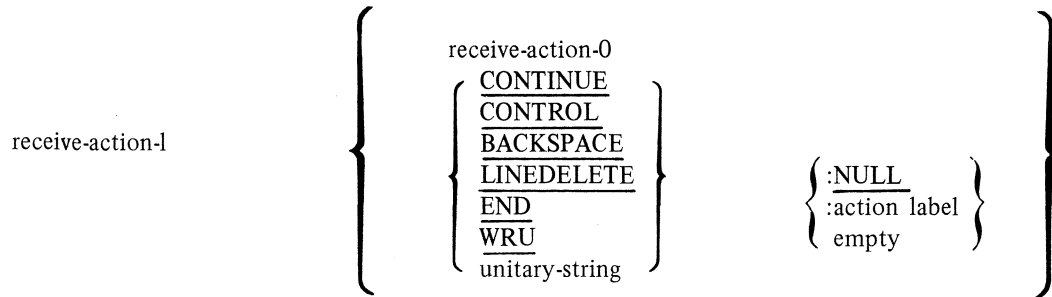
The following is a description of the receive-items and the specific action to be taken upon receipt of specific characters or occurrences of specific error conditions.

A receive-item is either a BCC-item or an ADDRESS-item (etc), and each is expressed in the following general format:



There are two formats for the receive-action part.



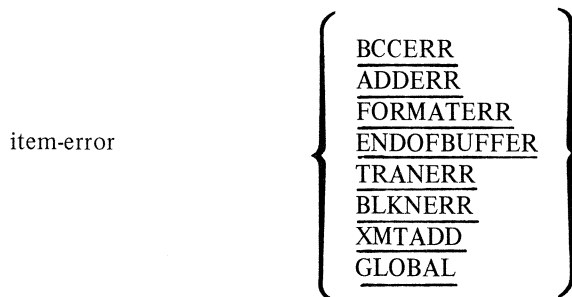


CONTROL, BACKSPACE, LINEDELETE, END and WRU are all indirect references to the Station and Terminal section declared values and cause a character-by-character comparison of each input character against each of the above named characters.

Unitary-string provides the ability to code a literal character within a RECEIVE statement causing a transfer out of the RECEIVE as described for CONTROL, WRU, etc.

CONTINUE allows the primary side of a duplex channel to “awaken” the auxiliary side out of a waiting type RECEIVE.

Item-error may only contain an item-error that corresponds to the receive items.



The ERROR designator in receive-action-0 is appropriate for all items; it specifies the action to be taken when the various error conditions occur. The presence of a receive-error-flag creates an error-switch that is unique to that RECEIVE statement. An error switch (refer to ERROR statement) and a receive-error-flag (PARITY, STOPBIT, etc) may not be used together in the same RECEIVE statement.

The following are various receive-item syntaxes.

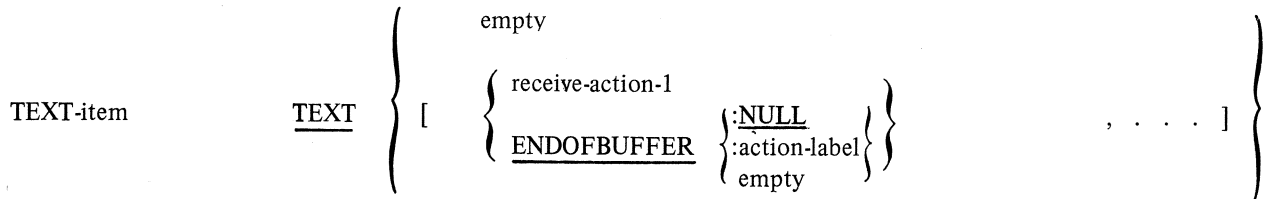
CHARACTER-Item

In this syntax, one character is received. Refer to the TEXT-item because all of the attributes for the text item apply to the CHARACTER-item, except ENDOFBUFFER.



TEXT-Item

Characters are received (and stored in the text part of the obtained message space) until an action item's action label results in a branch to a label, a terminate error, or a nonrecoverable error (such as a disconnect). If the occurrence of a character results in a branch outside the RECEIVE statement, that character is not stored automatically. Appropriate action items are text control characters, unitary-strings, and ENDOFBUFFERS. (An ENDOFBUFFER condition exists when too many characters have been received as specified by the terminal MAXINPUT statement.) The following explains these points and enumerates the special actions taken when text control characters are received.



Example

```
1:TALLY[2] = TALLY[2] + 1.
RECEIVE TEXT [END, WRU, BACKSPACE: 1]
TERMINATE NORMAL.
```

In this example, as each character is received by the communications subsystem, a check is made to determine if the character is the END, WRU, or BACKSPACE code for the station. If the character is END or WRU, an exit to the statement immediately following the RECEIVE statement is made.

If the BACKSPACE character is received, control is transferred to the statement labelled 1. If none of these three characters is found, the next character in the message is received, and the process continues.

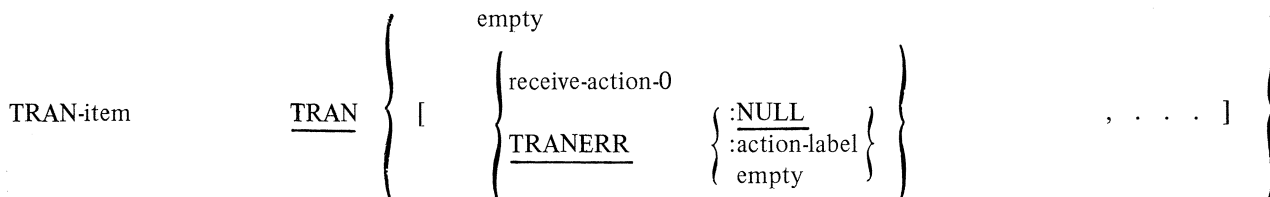
If the WRU, CONTROL, END, BACKSPACE, or LINEDELETE is used as action items, the system takes automatic action appropriate for the character, and then branches to the indicated label or the next statement depending on the action-label specified. The action taken for the WRU and CONTROL characters is to set the pertinent flag. The action for the END character is to do nothing except take the branch as specified by the action-label. (Note that the BACKSPACE and LINEDELETE characters are never stored in the text part of the message space, even if the action-label is “:NULL”.) Text may not be specified as an item in a control definition.

If a station has been declared as type bits, the last two characters received are assumed to be the CRC. (Refer to TYPE statement in Section 7.) These last two assumed CRC characters are compared against the calculated CRC. If at least two characters were not received, or if the assumed CRC did not validate, an INVALID FRAME error is generated causing a branch to the specified label.

RECEIVE

TRAN-Item

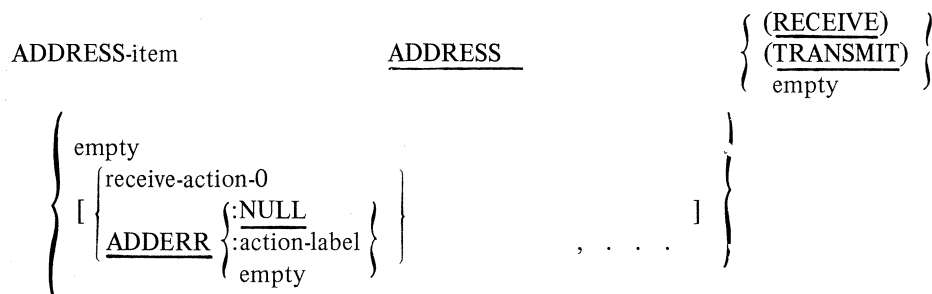
The proper number of transmission number characters are received and checked for agreement with the (receive) transmission number kept for the station with which the communications subsystem is communicating. If the numbers do not coincide, a transmission error (TRANERR) condition is obtained. The received transmission numbers are stored in the station table for subsequent transmission number functions. The transmission numbers are explicitly incremented by the NDL programmer. (The number of transmission number characters is specified in the station terminal description.) The transmission number is stored in the message header. TRAN may not be specified as an item in a control definition.



ADDRESS-Item

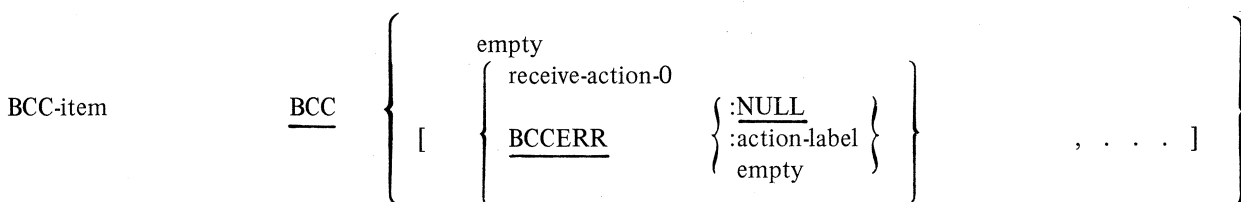
The proper number of receive or transmit address characters (the number of which is specified by the definition of the station terminal) are received and checked for agreement. Should the address characters not correspond, an address error condition results. If the receive action part has ADDERR action specified, the action-label is adhered to but otherwise an automatic TERMINATE NOLABEL is performed. (The analogous action occurs for all error conditions covered by item flags except CHARACTER and empty.)

This item is run-time conditional in control definitions.



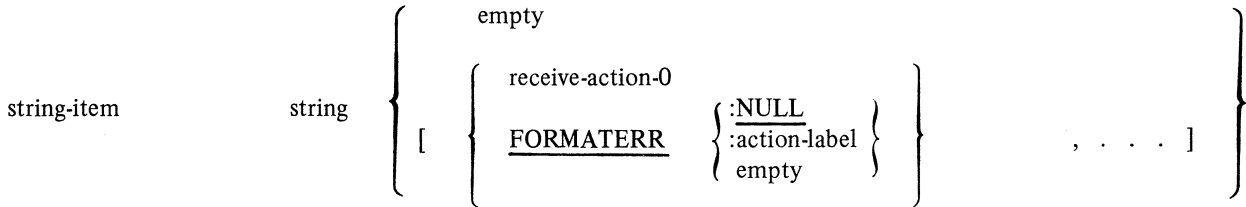
BCC-Item

A character is received and checked against the block check character which the communications subsystem has been accumulating during the processing of the request. BCCERR is appropriate for this item.



STRING-Item

Characters are received (and stored in the text string) and a check is made for correspondence, which if not found, results in a format error condition. FORMATERR is the item error flag for this item.

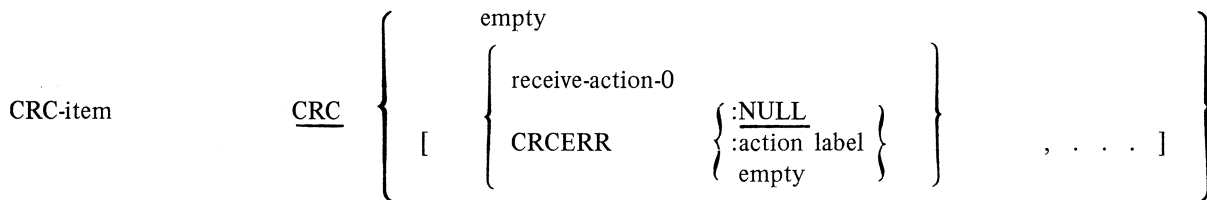


CRC-Item

Sixteen bits are received and checked against the CRC that has been locally computed. For a station not of type bits, RECEIVE CRC collects two characters from the line and then immediately performs validation. For a station of type bits, this instruction receives characters until it detects the flag sequence ending the frame. The last two characters received before the end flag are then used for CRC validation.

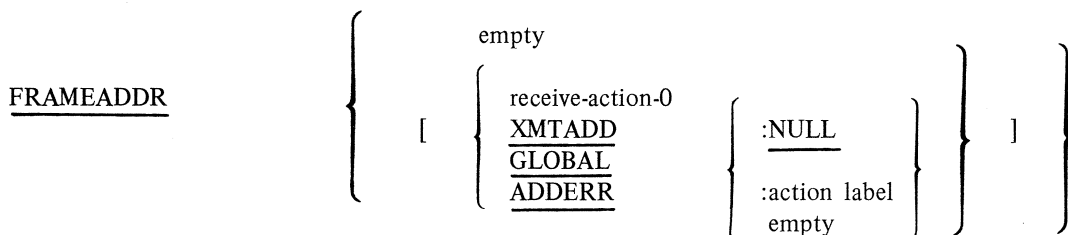
A branch to the receive-action-0 is performed if any status errors occur during the receive of the CRC. For a station of type bits, an INVALID FRAME occurs whenever more than two characters are received and the CRC validates.

If a CRC does not validate or if a bits station sees less than two characters during a RECEIVE CRC, then the CRCERR branch is performed.



FRAMEADDR-Item

This instruction conditions the DCS to receive a character and to compare it with the receive address character found in the station table. RECEIVE FRAME ADDRESS is used exclusively by bit-oriented line disciplines.



XMTADD and GLOBAL can be used only with RECEIVE FRAMEADDR. If the character received equals the GLOBAL ADDRESS (all ones), the GLOBAL BRANCH is taken. If the character equals the TRANSMIT ADDRESS, the XMTADD BRANCH is taken. If the character equals the RECEIVE ADDRESS, the next statement is executed; otherwise, the ADDERR BRANCH is taken. The RECEIVE FRAMEADDR can be used only if the terminal is defined as type bits. (Refer to TYPE statement in Section 6.)

Note 1

If NULL is used as an action-label in a RECEIVE TEXT statement, control returns to the RECEIVE statement after appropriate action is taken by the system.

If "empty" is used as an action-label, control is forced to the next sequential instruction.

If NULL is used as an action-label in any receive except RECEIVE TEXT statement, control is forced to the next sequential instruction.

Note 2

If a receive error flag, that was not specified in an error switch or the RECEIVE statement, is set or an unspecified item error flag is set, a TERMINATE NOLABEL occurs.

Examples

The ERROR switch designator is a grouping.

Example 1

```
ERROR [0] = STOPBIT:1,
           TIMEOUT:2,
           BREAK :3,
RECEIVE TEXT [ERROR[0] ].
RECEIVE TEXT [0].
```

Both of these RECEIVE statements perform the same function.

Example 2

```
RECEIVE (NULL) TEXT [0.
  ENDOFBUFFER:4,
  END,
  BACKSPACE:NULL,
  LINEDELETE:3,
  CONTROL:NULL,
  WRU].
```

Example 3

```
RECEIVE (1 SEC) EOT [ERROR[1],
  FORMATERR:4].
```

Example 4

```
RECEIVE [0, ETX:11, DEL:10].
RECEIVE CHAR [0, ETX:11, DEL:10].
```

Both of these RECEIVE statements perform the same function.

Example 5

```
RECEIVE (1 SEC) ADDRESS [ERROR[0]. ADDERR:21].
```

Example 6

```
RECEIVE (50 MILLI) [ERROR[0], PARITY].
```

This example shows an invalid RECEIVE statement. It utilizes an ERROR switch and a receive-error-item in the same RECEIVE statement.

Example 7

```
RECEIVE TEXT [1. ETX, ENDOFBUFFER:1, CONTROL:NULL].
```

This RECEIVE TEXT statement shows a check for an ETX (literal string) with no action-label. If an ETX is detected, a transfer of control to the next sequential statement occurs.

Example 8

```
RECEIVE TEXT [0, CONTINUE:17].  
RECEIVE [2, SOH:13, CONTINUE:22].
```

These statements result in a transfer out of the receive state if the primary side should execute a CONTINUE.

Example 9

```
RECEIVE TEXT [4, CONTINUE:NULL].  
17: RECEIVE [3], CONTINUE:17].
```

These statements are equivalent. If the primary executes a CONTINUE, control passes back to the receive.

Example 10

```
RECEIVE (500 MILLI) FRAMEADDR [2, XMTADD : 20,  
                                GLOBAL : 100,  
                                ADDERR : 125].
```

RESTORE

This statement is used exclusively with bit-oriented line disciplines. It allows the NDL programmer to move messages from the output save queue to the station queue.

The format for the statement is:

```
RESTORE {ALL}
```

This instruction removes either one message from the top (RESTORE) or all messages (RESTORE ALL) from the output save queue and queues with message(s) to the top of the station queue. The output save queue count is decremented by one or zeroed (if ALL is specified).

In addition, the toggle "EVENT1" is marked true in the header of that message space. If the output save queue count is zero when this instruction is executed, it is a NO-OP.

If the RESTORE statement is executed by the auxiliary half of a full-duplex pair, the instruction is a NO-OP.

The RESTORE statement can only be executed in a procedure used by a terminal of type bits. (Refer to the TYPE statement in Section 6.)

RETURN

This statement is used exclusively by bit-oriented line disciplines, and gives the NDL programmer the ability to remove messages from the output save queue.

The format for this statement is:

$$\underline{\text{RETURN}} \quad \left\{ \begin{array}{c} (\text{value}) \\ \underline{\text{ALL}} \end{array} \right\} \quad \left\{ \begin{array}{c} \underline{\text{NULL}} \\ \text{label} \end{array} \right\}$$

When using the RETURN statement (if <value> is specified), the number of messages is removed from the top of the output save queue and queued to the bottom of the result queue. The output queue count is decremented by the <value> which is a literal or is derived from the contents of a specified byte variable. If <value> is greater than the output save queue count, a branch to the specified label is executed, and no messages are removed from the output save queue. If <value> is zero, the instruction is a NO-OP.

If ALL is specified, all messages are removed from the output save queue, and are queued to the result queue. The output save queue count is zeroed. If the output save queue count is zero, the instruction is a NO-OP.

This statement is valid only in a procedure referenced by a terminal of type bits. (Refer to TYPE statement in Section 6.)

Examples

RETURN (TALLY [1]) [0]
RETURN (2) [1].

SHIFT

This statement allows the programmer to set the case of the character set. This statement is used to change the case of a character set to ensure current translation. An example of a two-level character set is the Baudot character set.

The format for the SHIFT statement is:

$$\underline{\text{SHIFT}} \quad \left\{ \begin{array}{c} \underline{\text{UP}} \\ \underline{\text{DOWN}} \\ \underline{\text{MIDDLE}} \end{array} \right\}$$

Examples

IF CHAR = FIGSHIFT THEN
SHIFT UP.
SHIFT MIDDLE.

Note

For automatic shifting, refer to Translation Table Syntax in Section 3.

STORE

Generally, this statement is used in receive request only. It takes a store item and stores it in the appropriate place. It is possible for an ENDOFBUFFER condition to occur when using this statement. If the specified MAXINPUT size is exceeded, the ENDOFBUFFER flag is set. The STORE TALLY, TOGS, or TOG may be used in an output request.

The format for the STORE statement is:

$$\underline{\text{STORE}} \quad \left\{ \begin{array}{c} \left\{ \begin{array}{c} \underline{\text{TOG}} [n] \\ \underline{\text{TALLY}} [0] \\ [1] \\ [2] \end{array} \right\} \\ \underline{\text{CHARACTER}} \\ \text{string} \\ \underline{\text{TOGS}} \\ \text{empty} \end{array} \right\} \quad \left\{ \begin{array}{c} \underline{\text{[ENDOFBUFFER:]}} \\ [label] \\ \underline{\text{[NULL]}} \\ \text{empty} \end{array} \right\} \quad \left\{ \begin{array}{c} \text{label} \\ \underline{\text{NULL}} \end{array} \right\}]$$

The STORE and STORE CHARACTER forms of this statement are equivalent.

An ENDOFBUFFER condition is an error condition. If empty is chosen as the action part, an automatic TERMINATE NOLABEL results. A NULL causes the program to transfer to the next statement in sequence and a label causes a branch to the specified label. The ENDOFBUFFER toggle is implicitly set true.

STORE CHAR

This version of the STORE statement stores the contents of the character register into the next sequential position in the text buffer. The text message pointer is incremented by one.

STORE String

This version of the STORE statement stores the designated string into the text buffer. The text message pointer is advanced by the length of the string.

STORE TOGS, TOG, or TALLY (0,1,2) provides a means of communicating with a MCS. The data is made available to the MCS.

STORE TALLY only allows the user to store the first three tallies (0,1,2). If a station is of type bits, TALLIES 3-18 cannot be stored. (Refer to TYPE statement in Section 7.)

Examples

```
010      RECEIVE CHAR [ERROR[0] ].
020      STORE CHAR [ENDOFBUFFER:2].
030      STORE [2] This line of code accomplishes the same
           thing as line 020.
040      STORE 4"2A" [NULL].
```

SUM

This statement either negates the horizontal summation (for BCC) for the previous character transmitted or received, or causes a character (for CRC) to be included in the CRC calculation.

The format for the SUM statement is:

$$\underline{\text{SUM}} \quad \left\{ \begin{array}{l} \text{unitary-string} \\ \text{CHARACTER} \\ \text{empty} \end{array} \right\}$$

For example, assume that a character has been received as part of a message and has been included in the horizontal block check character (BCC) being accumulated. The character may be eliminated from the BCC by the instruction SUM CHARACTER which causes the character contained in the character register to be exclusively ORed with the BCC, thus deleting the character from the partial sum.

The SUM and SUM CHARACTER forms of this statement are equivalent.

Examples

```
RECEIVE.
IF CHAR = STX THEN SUM.

RECEIVE.
IF CHAR = STX THEN SUM CHARACTER.
```


In both examples, the start-of-text character is eliminated from the BCC.

```
RECEIVE.  
IF CHAR = ETX THEN SUM 4"11".
```

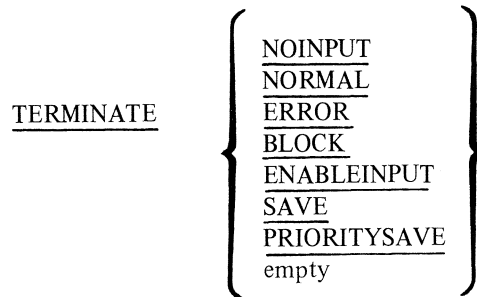
```
RECEIVE.  
IF CHAR = FLAG THEN  
  BEGIN  
    BINARY = FALSE  
    SUM  
    GO TO 27  
  END.
```

TERMINATE

This statement provides for exits and/or recognition of some event in a request procedure. This statement is not permitted in a line control procedure. TERMINATE statements may result in:

1. Control passing from the current request set to the appropriate line control procedure.
2. Control passing from an output request to an input request.
3. Recognizing the end of a block of input data.
4. Informing the system, upon recognition of an error, to create the proper result and transfer control to a line control procedure.

The format for the TERMINATE statement is:



TERMINATE NOINPUT

This statement indicates that an attempt to receive input from a station or transmit to a station was unsuccessful and no data transmission errors occurred. Control passes to the first executable statement in line control.

TERMINATE NORMAL

This statement indicates the satisfactory completion of an input (ENABLEINPUT) or output (REQUEST) function. Control then passes to the first executable statement of line control.

TERMINATE ERROR

This statement informs the system of a failure, completes a requested or enabled function, and inhibits initiation of any new functions for the station until the functions are requested by the system (MCS).

An error message is formulated to be returned to the host system (MCS error result). The station initiate queue is left intact (enable an MCS recall function to be performed) if the error occurred on an output request; only the first element of the queue is returned. The station status is marked as not ready. The appropriate line initiate routine is entered which idles a contention line but proceeds for other ready stations on a multi-point line.

The result flags and retry tally (which are copied from the current message) are placed into the error message and its header. If there is no current message (e.g., ENABLEINPUT), the result flags and retry tally

are copied from the station table, tally, toggles, message class, and line and station fields. Control is passed to the first statement of line control.

TERMINATE BLOCK

This statement for input is used to indicate a normal terminate for a block of information received from the station; subsequent blocks may follow. Control is immediately given back to the Receive Request section at the next executable statement that continues to process the next block of input from the station. On output, **TERMINATE BLOCK** is performed to ensure consecutive outputs to a station. Control is given to the next instruction in sequence when another block is available for output. The line is inactive if there is no block queued for output.

TERMINATE ENABLEINPUT

This statement provides an output request with the possibility of suspending itself in favor of the input request logic for the same station, as would be appropriate to some devices which may NAK the select if in the transmit ready mode.

If the station is input enabled, the current request and its message are held in the station initiate queue to be re-initiated later from the start, and the input logic is initiated; otherwise, control is immediately returned to the write request program at the next statement in sequence.

TERMINATE

The use of unqualified **TERMINATE** can be divided into two distinct functions, the half-duplex and full-duplex.

Full-Duplex

Since the primary side is dedicated to **TRANSMIT** and the auxiliary side to **RECEIVE**, each side has the ability to invoke the other (co line) to perform its specific function. When the co line has completed its function, an unqualified **TERMINATE** will return control to the top of the respective line control, that is, the primary line control when executed by the primary side, and the auxiliary line control when executed by the auxiliary side.

If message space is owned at the time the terminate is executed, in the case of the auxiliary side, this is discarded. In the case of the primary side, this is top queued to the station queue.

Half-Duplex

When executed in the **TRANSMIT** request, the current message is returned to the top of the station queue. When executed in the **RECEIVE** request and message space has been obtained (via **GETSPACE**), this space is placed on the top of the station queue. The next **INITIATE REQUEST** to be executed will find this input space at the top of the station and enter the **RECEIVE** request. This feature provides for temporary suspension of an input function.

TERMINATE NOLABEL

A **TERMINATE NOLABEL** statement results when an error occurs for which the programmer has no disposition. This error is reported to the MCS through an error result header. Control is passed to the beginning statement of line control.

TERMINATE SAVE

This statement is used to bottom queue (save) message space to the input save queue (when executed in the write request). The appropriate queue count (input save queue count or output save queue count) is incremented by one. Control passes to the line control procedure if the statement is executed in a write request. If executed in the read request, control passes to the next instruction.

TERMINATE PRIORITYSAVE

This statement is used to top queue message space to the input save queue (when executed in the read request) or the output save queue (when executed in the write request). The appropriate queue count is incre-

TIMER
TRANSMIT

mented by one (input or output save queue count). Control passes to the line control procedure (PRIMARY) if executed in a write request. If executed in a read request procedure, control passes to the next instruction.

TIMER

This statement provides the ability to vary the time given over to manager options.

The format for this statement is:

TIMER = < time >

This statement loads a GROSS TIMER with 2's complement of the stated time. The noresponse reference toggle is reset on execution of this statement. Once primed, the GROSS TIME indicates a timeout by marking the noresponse true.

If the time value is zero, the noresponse toggle is reset and the GROSS TIMER is disabled. The maximum time allowed is 65 seconds.

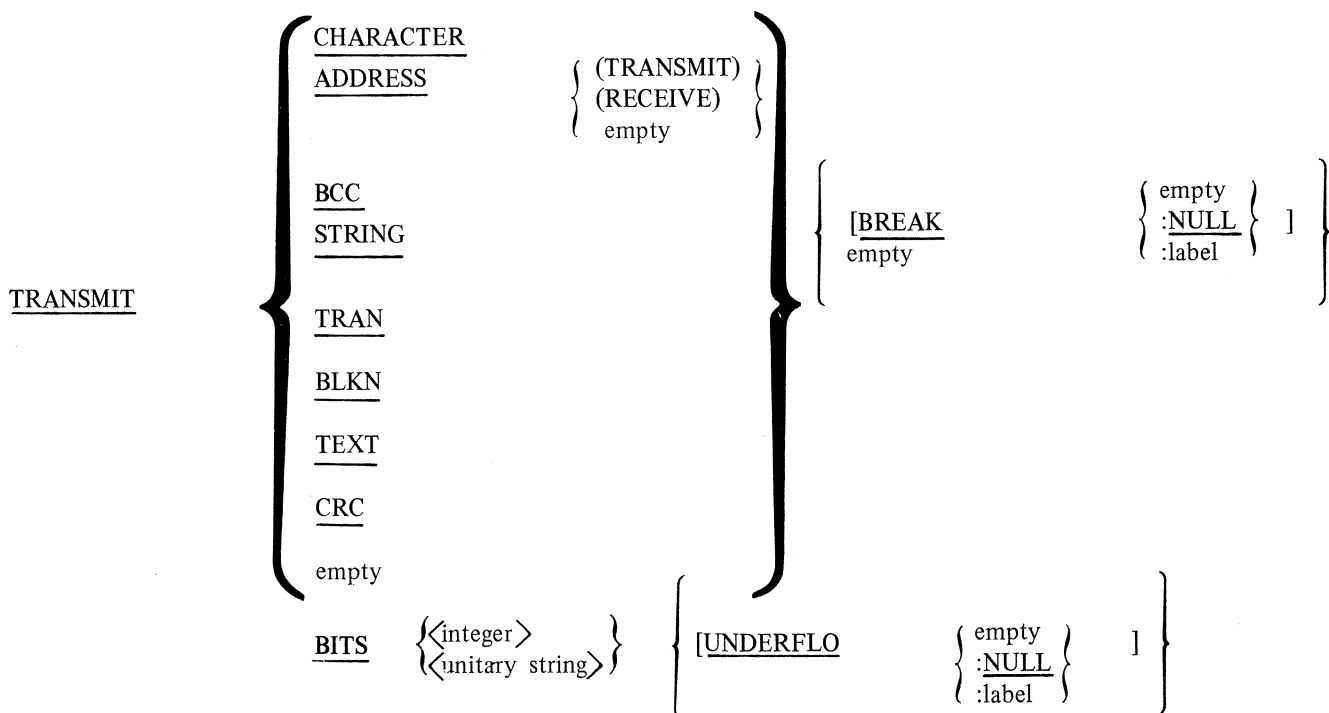
Both halves of a full-duplex pair can reprime the GROSS TIMER. Both can test or alter the noresponse toggle.

Example

TIME = 10 MILLI.
TIMER = 1 SEC.

TRANSMIT

This statement is used for output operations to a station. When the TRANSMIT statement appears in a line control definition, TRAN and TEXT are not allowed.



If the TRANSMIT empty format is used, the current contents of the character register are transmitted; otherwise, the item specified indicates the value to be loaded into the character register before transmission. The ADDRESS, TRAN, and BCC forms cause the station address, transmission number, and block-check-character, respectively, to be loaded and transmitted down the line. The CRC form causes the 16-bit CRC value to be transmitted.

The BITS form is used exclusively with bit-oriented line disciplines. When executed, the instruction causes the specified eight-bit <integer> or <unitary string> to be transmitted "as is", with the option selected to inhibit zero insertion. Translation is not performed on the <integer> or <unitary string>, nor is the <integer> or <unitary string> included in the CRC calculation.

If a synchronous underflow is detected while executing this instruction, a branch to the specified UNDERFLOW ADDRESS is performed.

"TRANSMIT string" loads and transmits each character of the string specified until the string is exhausted. The transmit action part specifies the action strobe taken upon detection of a break. If it is empty, then an automatic TERMINATE NOLABEL results from a break; otherwise, an empty label signifies a branch to the next statement. A :NULL label signifies that execution should proceed as if the condition associated with the label did not occur. A :label specifies that a branch to the designated label should occur. An empty ADDRESS qualifier defaults to the receive address.

The TRANSMIT and TRANSMIT CHAR forms are equivalent. The statement TRANSMIT TEXT is equivalent to:

```

FETCH [ENDOFBUFFER :2]
TRANSMIT,
GO TO 1.
    
```

Examples

1. TRANSMIT EOT.
TRANSMIT ADDRESS.
TRANSMIT SEL.
TRANSMIT ENQ.
FINISH TRANSMIT.
2. TRANSMIT BITS 65 [UNDERFLO:1].

WAIT

This statement is used in conjunction with the CONTINUE statement. WAIT provides a way for one side of a duplex channel to synchronize activity with the other. WAIT causes suspension of code execution on the line which executes it.

$$\text{WAIT} \left\{ \begin{array}{l} (\text{INTEGER} \\ \text{empty} \end{array} \left\{ \begin{array}{l} \frac{\text{MICRO}}{\text{MILLI}} \\ \frac{\text{SEC}}{\text{MIN}} \end{array} \right\} \left\{ \begin{array}{l} \text{:action label} \\ \text{empty} \end{array} \right\} \right\}$$

WAIT may be used in both control and request sets.

Examples

1. WAIT.
2. WAIT (100 MILLI).
3. WAIT (1 SEC:24).

Example 1 illustrates an unconditional suspension that requires the “other” side of a duplex channel to execute a CONTINUE in order to free the WAITing side.

Example 2 causes the line executing the WAIT to continue the next instruction if: 1) the “other” side executes a CONTINUE; or 2) the expressed time value expires. This type of WAIT might be illustrated in part of an output request which contains code executed by both the primary and auxiliary sides. The primary is executing from the first line of the example.

```
IF NOT AUX (LINE(BUSY)) THEN
BEGIN
  FORK 105.
  WAIT.
  IF LINE (TOG[1]) THEN
    TERMINATE NORMAL
  ELSE TERMINATE ERROR.
END.
```

```
LINE (TALLY[0] = 1 + LINE (TALLY[0])).
LINE (BUSY) = FALSE
WAIT (100 MILLI).
LINE (BUSY) = TRUE
TERMINATE.
```

```
105.  INITIATE RECEIVE. % AUX PORTION
      RECEIVE ACK [ERROR[2], FORMATERR: 109].
      IF LINE (TALLY[0]) NEQ 0 THEN
        LINE (TALLY[0]) = LINE (TALLY[0]) -1.

      LINE (TOG[1]) = TRUE.
106.  CONTINUE
      TERMINATE.
109.  LINE (TOG[1]) = FALSE.
      GO 106.
```

LINE (TOG[1]) is used as a signal between sides. LINE (TALLY [0]) contains a count of output messages for which a response has not yet been received. These conventions are not exhausted, nor is the response code elaborated to treat numbered responses from separate stations.

Example 3 illustrates a conditional WAIT in which, if the “other” side executes a CONTINUE before the time limit expires, execution proceeds at the next instruction. If the time limit expires, execution resumes by the WAITing side at the declared label.

SECTION 5

MODEM SECTION

GENERAL

This section provides information to the communications subsystem to enable proper communication between it and its attached modems/data sets. The pertinent characteristics of each type of modem/data set are described in the statements contained in this section.

MODEM SECTION DESCRIPTION

The statements that allow these characteristics to be entered into the system are shown below and described on the following pages. All statements end with a period.

The format for the Modem section is:

MODEM identifier {
 SPEED = integer.
 TRANSMITDELAY = time.
 TYPE = type, options.
 NOISEDELAY = time.
 { LOSSOF CARRIER = DISCONNECT. }
 empty } . . .

For each type of modem/data set on the system there must exist some variation of the above syntax.

LOSSOFCARRIER

The LOSSOFCARRIER statement is used only for a switched network (dial up). It indicates the action to be taken when LOSSOFCARRIER is detected by the communications subsystem. When a modem is declared with this statement, special logic is invoked in addition to the normal Modem section code.

The format for the LOSSOFCARRIER statement is:

LOSSOFCARRIER = DISCONNECT

If this statement is included, the line will disconnect if LOSSOFCARRIER is detected. If not, a persistent LOSSOFCARRIER causes the line to go not ready without disconnecting it.

A terminal initiated disconnect (unexpected disconnect) is determined under normal switched logic by carrier detected false. This is construed as an unexpected disconnect.

Example

LOSSOFCARRIER = DISCONNECT.

NOISEDELAY

This statement defines the amount of time delay needed to compensate for the noise on the line to which the modem is connected. For INITIATE RECEIVE, the value of the line modem is used. The NOISEDELAY statement is required.

The format for the NOISEDELAY statement is:

NOISEDELAY = integer $\left\{ \begin{array}{l} \text{MICRO} \\ \text{MILLI} \\ \text{SEC} \\ \text{MIN} \end{array} \right\}$

The time must be defined as an integer in microseconds, milliseconds, seconds, or minutes. If a zero (0) is specified as the transmit delay value (time), the time unit specifier need not be represented.

Examples

NOISEDELAY = 0.
NOISEDELAY = 10 MILLI.

Note

The NOISEDELAY should be non-zero only when the modem is connected in a half-duplex mode of operation.

SPEED

This statement is used in conjunction with the modem TYPE statement. If an asynchronous type is specified, the integer is used to represent the maximum baud rate at which the modem may operate. If the modem is synchronous, the stated speed is inferred to be that baud rate at which the modem will operate.

The format for the SPEED statement is:

SPEED = integer

Example

SPEED = 9600.

The allowable speed value is:

ASYNC	SYNC
50	600
75	1200
100	2000
110	2400
150	3600
200	4800
300	7200
600	9600
1200	
1800	
2400	
4800	
9600	
19200	
38400	

If neither the RATESELECT nor STANDBY options are specified in the modem TYPE statement, only one speed may be specified.

TRANSMITDELAY

This statement defines the amount of time required for the data set to go from a request-to-send status to a ready-to-send status. The TRANSMITDELAY statement is required.

The format for the modem TRANSMITDELAY statement is:

$$\text{TRANSMITDELAY} = \text{integer} \left\{ \begin{array}{l} \text{MICRO} \\ \text{MILLI} \\ \text{SEC} \\ \text{MIN} \end{array} \right\}$$

The time must be defined in microseconds, milliseconds, seconds, or minutes.

If zero (0) is specified as the transmit delay value, the time unit specifier need not be represented.

Example

TRANSMITDELAY = 250 MILLI.

TYPE

This statement is used to define the modem's capability to the system. The DIALOUT option specifies that the modem itself has dialout capability.

The format for the TYPE statement is:

SECTION 6

TERMINAL SECTION

GENERAL

This section provides a description of the physical aspects of different types of terminals that may be on the data communications system. This description is required so that the communications subsystem may handle communications with each type of device.

TERMINAL DEFINITION DESCRIPTION

For each type of terminal with different physical aspects, a terminal specification must exist. For terminals with like characteristics, one terminal specification is required.

The general format for a terminal definition is:

<u>TERMINAL</u>	{	<u>DEFAULT</u>	}	terminal- identifier:	{	terminal-attribute-1. terminal-attribute-2. . . . terminal-attribute-n. empty	}	...
-----------------	---	----------------	---	--------------------------	---	---	---	-----

If DEFAULT is specified, the compiler recognizes that the terminal attributes specified are to be associated with the terminal identifier specified. When the terminal attribute DEFAULT is used, the compiler refers to the list of terminal attributes designated by that terminal identifier. A default terminal may not be designated as part of a station.

The various terminal attribute statements are discussed in the following pages. All statements end with a period.

ADDRESS

This statement specifies the number of characters that are used to address a station associated with a terminal type. The ADDRESS statement is required.

The format for the ADDRESS statement is:

$$\underline{\text{ADDRESS}} = \left(\begin{array}{c} \left(\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} \right) \\ \text{NULL} \end{array} \right) \left(\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ \text{empty} \end{array} \right)$$

A device having a receive address and a transmit address is possible. The first to be specified is the number of characters that make up the receive address (1, 2, 3). The second is the number of characters that make up the transmit address. If NULL or 0 is specified, an address does not exist and the address length is not required.

If the transmit address is the same length as the receive address, the length need not be specified. If the terminal is type bits, the receive address size must be 1. The transmit address size must be 1 or 0. (Refer to TYPE statement in Section 6.)

Examples

ADDRESS = 2. % BIDS
ADDRESS = NULL. % CONTENTION
ADDRESS = 2.
ADDRESS = 0.

BACKSPACE

This statement defines the character that is to be recognized as the backspace character for the associated terminal.

The format for the BACKSPACE statement is:

$$\underline{\text{BACKSPACE}} = \text{unitary-string}$$

The unitary-string may be a hexadecimal or ASCII value, or a constant that is declared in the Constant section. The backspace character causes a backspace; a delete of the preceding character results if a check for backspace is in the RECEIVE statement.

Examples

BACKSPACE = BS.
BACKSPACE = 4"16".

Note

This character must be specified if it is referenced in the terminal's request or control procedures.

BLOCKED

This statement, if TRUE, informs the communications subsystem that the terminal is capable of sending and/or receiving data in blocked format.

The format for the BLOCKED statement is:

$$\left\{ \begin{array}{c} \underline{\text{BLOCK}} \\ \underline{\text{BLOCKED}} \end{array} \right\} = \left\{ \begin{array}{c} \underline{\text{TRUE}} \\ \underline{\text{FALSE}} \end{array} \right\}$$

The request procedure defined for the terminal should be coded in a manner that allows blocked format.

Examples

BLOCKED = FALSE.
BLOCK = TRUE.

BYTE

This statement describes the transmission byte to the number of data bits contained and whether or not a parity bit exists. The BYTE statement is required.

The format for the BYTE statement is:

$$\underline{\text{BYTE}} = \text{integer} \quad \left\{ \begin{array}{l} \text{, PARITY} \\ \text{empty} \end{array} \right\}$$

The integer specifies the number of data bits in the transmitted character. The values for integer are: 4, 5, 6, 7, and 8. Parity may not be specified if a value of 8 is used. Parity must be specified if a value of 4 is used.

The parity action must be consistent with the terminal PARITY statement. That is, if odd or even vertical parity is declared, the BYTE PARTIY option must be specified. The parity bit may be declared present even though it is not used (vertical parity is null) as it is relevant for teletypes and similar devices. All terminals on a line must have the same byte size. If the terminal is type bits, then the integer that specifies the number of data bits must equal 8. (Refer to TYPE statement in Section 6.)

Example

BYTE = 7, PARITY.

CARRIAGE

This statement defines the character that is to be recognized as the carriage return character for the associated terminal.

The format for the CARRIAGE statement is:

$$\underline{\text{CARRIAGE}} = \text{Unitary-string}$$

The unitary-string may be a hexadecimal or ASCII value, or a constant that is declared in the Constant section.

Examples

CARRIAGE = CR.
CARRIAGE = 4 "OD".

Note

Unless declared as an NDL constant, this data character is available only to the MCS.

CLEAR

This statement defines the clear character for the associated terminal.

The format for the CLEAR statement is:

$$\underline{\text{CLEAR}} = \text{unitary-string}$$

The clear character normally causes the terminal to execute a combined home and clear function by moving the cursor to the home position and erasing all data. For the clear character to be recognized, the SCREEN statement must be set to TRUE.

The unitary-string may be a hexadecimal or ASCII value, or a constant that is declared in the Constant section.

Examples

CLEAR = FF.
CLEAR = 4 "OC".

Note

Unless declared as an NDL constant, this data character is available only to the MCS.

CODE

This statement specifies the character set that is used to communicate with a specific terminal type. The CODE statement is required.

The format for the CODE statement is:

CODE = { BAUDOT
BCL
ASC67
EBCDIC
BINARY
Translate-table-id }

Translation of input to the communications subsystem from a terminal is always ASCII. If the terminal is type bits, then code must be EBCDIC on ASC67. (Refer to TYPE statement in Section 6.)

Examples

CODE = ASC67.

CONTROL

This statement designates the name of the line control procedure to be used on the line to which the specified terminal is attached. The control statement is required.

The format for the CONTROL statement is:

CONTROL = control-identifier-1 { ,control-identifier-2 }
empty }

All terminals on a line must specify the same line control procedure.

Examples

CONTROL = POLL. IDLE.
CONTROL = CONTENTION.

If two control procedures are specified, the terminal must be full duplex. The second procedure specifies the control procedure to be used on the auxiliary line, while the first control is designated on the primary.

If a full duplex terminal does not have two control procedures specified, a dummy procedure containing an IDLE only is created.

DEFAULT

This statement specifies the identifier associated with a set of predefined default terminal attributes. These attributes are assigned to the terminal presently being defined.

The format for the DEFAULT statement is:

DEFAULT = terminal-identifier

Terminal-identifier refers to the terminal-identifier of the default terminal-definition in which the desired terminal attributes exist. If the DEFAULT statement is used, it must be the first statement in the definition.

Example

DEFAULT = TCTYPE.

END

This statement defines the character to be recognized, if referenced in a RECEIVE statement, as the ending (terminating) character for a transmission from the associated terminal.

The format for the END statement is:

END = unitary-string.

The unitary-string may be an ASCII or hexadecimal value, or a constant that is declared in the Constant section.

Examples

END = ETX.

END = 4'03'.

Note

This character must be specified if it is referenced in the terminal's request or control procedures.

HOME

This statement defines the home character for the associated terminal.

The format for the HOME statement is:

HOME = unitary-string.

The home character normally causes the terminal to execute a cursor home function by moving the cursor to the first position on the first line.

The unitary-string may be a hexadecimal or ASCII value, or a constant that is declared in the Constant section.

Examples

HOME = DC4.

HOME = 4'14'.

Note

Unless declared as an NDL constant, this data character is available only to the MCS.

LINEDELETE

If referenced in a RECEIVE statement, this statement defines the character to be recognized as the line delete character for the associated terminal.

The format for the LINEDELETE statement is:

LINEDELETE = unitary-string

The unitary-string may be a hexadecimal or ASCII value, or a constant that is declared in the Constant section.

Examples

LINEDELETE = DC1.
LINEDELETE = 4"11".

Note

This character must be specified if it is referenced in the terminal's request or control procedures.

LINEFEED

This statement defines the line feed character for the associated terminal.

The format for the LINEFEED statement is:

LINEFEED = unitary-string.

The unitary-string may be a hexadecimal or ASCII value, or a constant that is declared in the Constant section.

Examples

LINEFEED = LF.
LINEFEED = 4"0A".

Note

Unless declared as an NDL constant, this data character is available only to the MCS.

MAXINPUT

This statement applies only to inputs from the terminal and defines the maximum size allowable. The MAXINPUT statement is required.

The format for the MAXINPUT statement is:

MAXINPUT = integer

Examples

MAXINPUT = 80
MAXINPUT = 255.

MOD

This statement determines the range of the sequence field during bit-oriented communications. The sequence range is from 0 through MOD value - 1.

The format for the MOD statement is:

MOD = { 8. }
 { 128. }

Examples

SAVE = 6.

SCREEN

This statement is set TRUE only if the associated terminal has display capabilities.

The format for the SCREEN statement is:

$$\underline{\text{SCREEN}} = \left\{ \begin{array}{l} \underline{\text{TRUE}} \\ \underline{\text{FALSE}} \end{array} \right\}$$

Example

SCREEN = TRUE.

Note

This data is available only to the MCS.

SPEED

This statement defines the line speed(s) possible for that terminal. This statement is required.

$$\underline{\text{SPEED}} = \left\{ \begin{array}{l} \text{integer} \left\{ \begin{array}{l} (1) \\ (2) \\ \text{empty} \end{array} \right\} \left\{ \begin{array}{l} \text{empty} \end{array} \right\} \end{array} \right\} \dots$$

If the terminal type is asynchronous, the terminal SPEED statement specifies up to five speeds at which the terminal may operate. If more than one speed is listed, then stations whose terminal statement names such a terminal must use the station SPEED statement to specify one of the listed speeds. No STOPBIT option is allowed. Also, the STOPBIT option may be empty and, as such, implies one stop bit.

If the terminal TYPE is synchronous, one maximum speed is specified.

Examples

SPEED = 2400.

SPEED = 9600 (1).

Note

Possible asynchronous speeds are: 75, 100, 110, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, and 38400.

Possible synchronous speeds are: 1200, 2000, 2400, 3600, 4800, 7200, and 9600.

TALLIES

This statement specifies whether or not the control and request procedures used by this terminal may reference tally [3] through tally [18].

The format for the TALLIES statement is:

$$\underline{\text{TALLIES}} = \left\{ \begin{array}{l} \underline{\text{TRUE}} \\ \underline{\text{FALSE}} \end{array} \right\}$$

If TALLIES is set TRUE, the additional tallies (3-18) may be referenced; if FALSE, the tallies may not be referenced. The default value is FALSE.

The TALLIES statement must agree with the TALLIES statement for all stations that use the same terminal. (Refer to TALLIES statement in Section 7.)

Examples

TALLIES = TRUE
TALLIES = FALSE % DOCUMENTATIONAL

TIMEOUT

This statement defines the time period: 1) between the receipt of one character (indicated by the last stop bit of that character for asynchronous lines) and the start of the next character (indicated by the start bit of that character for asynchronous lines) from a terminal; or 2) between the initiation of a read and the receipt of the first character from the station. The TIMEOUT statement is required.

$$\underline{\text{TIMEOUT}} = \text{integer} \left\{ \begin{array}{c} \underline{\text{MICRO}} \\ \underline{\text{MILLI}} \\ \underline{\text{SEC}} \\ \underline{\text{MIN}} \end{array} \right\}$$

The time must be defined as an integer of microseconds, milliseconds, seconds, or minutes. This time may also be used to specify the time period allowed for a response to an output.

Examples

TIMEOUT = 10 MILLI.
TIMEOUT = 3 SEC.

TRANSMISSION

This statement defines the length of the transmission number for the associated terminal.

The format for the TRANSMISSION statement is:

$$\underline{\text{TRANSMISSION}} = \left\{ \begin{array}{c} \underline{\text{NULL}} \\ \underline{0} \\ \underline{1} \\ \underline{2} \\ \underline{3} \end{array} \right\}$$

The number zero and NULL are equivalent and specify that no transmission number is to be used. NULL may not be used if the item TRAN is mentioned in a request procedure of the terminal. If the terminal is type bits, the transmission statement is disallowed. (Refer to TYPE statement in Section 6.)

Examples

TRANSMISSION = NULL.
TRANSMISSION = 2.

TRANSPARENT

This statement allows an MCS to know, by a communicate, that a terminal is capable of sending or receiving transparent data.

The format of the TRANSPARENT statement is:

$$\underline{\text{TRANSPARENT}} = \left\{ \begin{array}{c} \underline{\text{TRUE}} \\ \underline{\text{FALSE}} \end{array} \right\}$$

The TRANSPARENT statement is only valid on a terminal specified as type bits. (Refer to TYPE statement in Section 6.)

Example

TRANSPARENT = TRUE.

TURNAROUND

This statement specifies the time between the communications subsystem's "turning the line around" and the terminal's being able to receive characters. The TURNAROUND statement is required.

The format for the TURNAROUND statement is:

$$\underline{\text{TURNAROUND}} = \text{integer} \left\{ \begin{array}{l} \underline{\text{MICRO}} \\ \underline{\text{MILLI}} \\ \underline{\text{SEC}} \\ \underline{\text{MIN}} \end{array} \right\}$$

The time specified should be the time from the communications subsystem's initiating transmit to the time the terminal is capable of receiving characters. It is an integer value of the number of microseconds, milliseconds, seconds, or minutes that represent the proper turnaround time.

Examples

TURNAROUND = 250 MILLI.
TURNAROUND = 0.

Note

If a TURNAROUND of 0 is specified, the time unit descriptor need not be present.

TYPE

This statement provides for the description of the type of connection for this terminal. Several may be specified indicating several possibilities for each station.

The format for the TYPE statement is:

$$\underline{\text{TYPE}} = \left\{ \begin{array}{l} \underline{\text{ASYNC}} \left\{ \begin{array}{l} \text{empty} \\ \left\{ \begin{array}{l} \underline{\text{MODEM}} \\ \underline{\text{BDI}} \\ \underline{\text{DIRECT}} \\ \underline{\text{TELEX}} \end{array} \right\} \\ \dots \end{array} \right\} \left\{ \begin{array}{l} \underline{\text{.DUPLEX}} \\ \text{empty} \end{array} \right\} \\ \underline{\text{SYNC}} \left\{ \begin{array}{l} \underline{\text{MODEM}} \\ \left\{ \begin{array}{l} \underline{\text{BITS}} \\ \underline{\text{.DUPLEX}} \end{array} \right\} \\ \underline{\text{WRAPAROUND}} \\ \text{empty} \end{array} \right\} \end{array} \right\}$$

MODEM, DIRECT, BDI, and TELEX qualify the ASYNC type. Several may be specified indicating that there are several possibilities for each relevant station. If MODEM, DIRECT, or BDI are not specified, all are inferred as possibilities. In these cases, the relevant station declarations must contain a station TYPE statement in order to select one of the possible ASYNC types.

The WRAPAROUND option specifies that a full-duplex line adapter is capable of an internal wrap-around; that is, all output is wrapped back as input on the auxiliary side of the line. Systems whose adapters do not have this capability will ignore the option and attempt to run the terminal.

If the terminal is declared as SYNC, the MODEM option must be present for these types. The terminal TYPE statement is required.

The terminal DUPLEX option must be consistent with the read and write requests and line control(s) declared for the terminal.

TELEX excludes DIRECT and BDI, but implies MODEM.

BDI and MODEM may be specified with DUPLEX, but DIRECT connect is disallowed.

A BDI non-duplex terminal cannot be used on a BDI DUPLEX line.

If a terminal is non-telex and the line specifies TELEX, DUPLEX is not allowed in the terminal specification. If a terminal specifies TELEX, the station and line must specify TELEX.

If a TELEX is specified:

1. The byte statement equals 5 with no parity.
2. Parity must specify null.
3. Speed is equal to 50, and only one speed is specified.

Examples

The BITS option indicates that the terminal is capable of using bit-oriented line disciplines (i.e., BDLC, SDLC, HDLC). The BITS option may be used with SYNC type terminals only.

If BITS is declared, then:

1. The terminal must be SYNC and MODEM.
2. The ADDRESS statement (refer to ADDRESS in Section 6) must be "1, 1" or "1,0".
3. BYTE must be 8. (Refer to BYTE statement in Section 6.)
4. CODE can only be EBCDIC or ASC67. (Refer to CODE statement in Section 6.)
5. PARITY must be HORIZONTAL : CRC (ECMA). (Refer to PARITY statement in Section 6.)
6. SAVE and MOD statements are required. (Refer to these statements in Section 6.)

Examples

TYPE = ASYNC (DIRECT).
TYPE = SYNC (MODEM), DUPLEX.
TYPE = ASYNC (MODEM, BDI), DUPLEX
TYPE = SYNC (MODEM), DUPLEX, BITS.
TYPE = SYNC (MODEM), BITS.

WRAPAROUND

This statement specifies whether automatic WRAPAROUND occurs for messages that exceed the value specified in the LINEWIDTH statement.

The format for the WRAPAROUND statement is:

WRAPAROUND = $\left\{ \begin{array}{l} \text{TRUE} \\ \text{FALSE} \end{array} \right\}$

If true, WRAPAROUND occurs; that is, if the device receives a message that exceeds the line width, it automatically places the excess on successive lines.

Example

WRAPAROUND = TRUE.

Note

This data is available only to the MCS.

WIDTH

This statement defines the length of a physical line of output from the terminal. This length must be less than or equal to the length defined in the MAXINPUT statement.

The format for the WIDTH statement is:

WIDTH = integer.

A line of output may be less than the number of characters specified by the WIDTH statement. If WIDTH is not defined, a default value of 255 is used.

Example

WIDTH = 72.

Note

This data is available only to the MCS.

WRU

If referenced in a RECEIVE statement, this statement defines the character that is to be recognized as the WRU (WHO ARE YOU) character for the associated terminal.

The format for the WRU statement is:

WRU = unitary-string.

The unitary-string may be an ASCII or hexadecimal value, or a constant that was declared in the Constant section.

Examples

WRU = ENQ.

WRU = 4 "05".

Note

This character must be specified if it is referenced in the terminal's request or control procedures.

SECTION 7

STATION SECTION

GENERAL

This section specifies the logical characteristics of the remote stations to the data communications network. The type of terminal, the station's address, and the modification of standard terminal type information (such as end-of-message character, and line width) are some of the station information that is specified in the Station section.

STATION DEFINITION DESCRIPTION

For each station on the data communications network, a station definition must exist. Stations with like characteristics may use the station DEFAULT statement to reduce source code.

The general format for a station definition is:

<u>STATION</u>	$\left\{ \begin{array}{l} \text{DEFAULT} \\ \text{empty} \end{array} \right\}$	station- identifier:	$\left\{ \begin{array}{l} \text{station-attribute-statement-1} \dots \\ \text{station-attribute-statement 2} \dots \\ \text{station-attribute-statement 3} \dots \end{array} \right\}$
----------------	--	-------------------------	--

If DEFAULT is specified, the compiler recognizes that the specified station-attribute-statements are to be associated with the specified station-identifier. When the station-attribute-statement DEFAULT is specified, the compiler uses the station-identifier of the DEFAULT statement to decide with which station the DEFAULT definition is to be associated. A STATION DEFAULT cannot be designated in a Line, or File section.

Examples

```
STATION DEFAULT DCTC5:  
  TERMINAL = TC51A.  
  ADDRESS = "1A".
```

The DEFAULT statement further explains the use of the station format. All statements end with a period.

ADDRESS

This statement specifies the address of a station for operations such as polling and selecting.

The format for the ADDRESS statement is:

$$\text{ADDRESS} = \left\{ \begin{array}{l} \text{address - string} \\ (\text{receive - address-string, transmit-address-string} \\ (\quad * \quad \quad \quad , \text{transmit-address-string}) \\ (\text{receive-address-string,} \quad \quad \quad * \quad) \end{array} \right\}$$

If the ADDRESS statement is not used in a station definition, the mode of the associated station must be contention. The length of the address specified must be equal to the length defined in the terminal ADDRESS statement for the respective terminal type.

If the terminal specified a non-zero address length, the station must specify an address.

The "*" indicates that no address is required. If only one address is specified, both the receive and transmit address are the same.

The address-string may be either an ASCII or hexadecimal string.

Examples

```
ADDRESS = "1A". % CHARACTER STRING
ADDRESS = "J". % CHARACTER STRING
ADDRESS = `4"F1C1". % HEX STRING
ADDRESS = 4 "D1". % HEX STRING
ADDRESS = ("1J", "J12")
ADDRESS = (4"F1D1", 4"D1F1F2")
ADDRESS = (4"01", *).
```

Note

If the specified terminal has a non-zero address length specified in the Terminal section, the Station section must specify an address.

CONTROL

This statement defines a character which is referenced in a RECEIVE statement and is used as the first nonblank character in a message text. This character must be specified if CONTROL or CONTROLFLAG is referenced in the control or request procedures.

The format for the CONTROL CHARACTER statement is:

$$\text{CONTROL} = \text{unitary-string}$$

The character in the unitary-string may be expressed as either an ASCII or hexadecimal value.

Examples

```
CONTROL = "$".
CONTROL = 4"5B". %HEX
```

DEFAULT

This statement specifies the name of a set of station-attribute-statements to be used for a station. For stations with like characteristics, it is advantageous to group these common characteristics under a DEFAULT

definition and list any remaining statements under each individual station definition. The NDL compiler then refers to the default list to complete the station specification.

The format for the DEFAULT statement is:

DEFAULT = station-default-identifier.

The station-default-identifier must have been described previously in a station DEFAULT definition. If the DEFAULT statement is used, it must be the first statement in the definition.

Example

```
STATION DC100RJE:  
  DEFAULT = DC100RJES.  
  MODEM = M201B.
```

ENABLEINPUT

If set true, this statement implies that input from a station on an attached line will be accepted and/or solicited (polled devices) upon completion of communications subsystem initialization without a specific request from an MCS.

The format for the ENABLEINPUT statement is:

ENABLEINPUT = { TRUE }
 { FALSE }

Example

```
ENABLEINPUT = FALSE.
```

Note

If ENABLEINPUT = TRUE, the station's MYUSE statement must specify input.

FREQUENCY

This statement specifies the approximate intervals in arbitrary units at which the station being described is to be polled.

The format for the FREQUENCY statement is:

FREQUENCY = integer.

The maximum number that may be specified is 255. This value can be interrogated in the Line Control section and the Request section; it is a read-only value.

Example

```
FREQUENCY = 75.
```

INITIALIZE

This statement provides the ability to preset user tallies (0, 1, and 2) and toggles (7 through 0) to values that apply at data comm initialization time. Tallies and toggles not initialized are set to zero or false, respectively.

INITIALIZE { TOG[n]
 TALLY[n]
 TALLY[n] [bit-n]
 TOGS } = { TRUE
 FALSE
 unitary-string
 integer } , . . .

TOG[n] and TALLY [n] [bit-n] must be assigned TRUE or FALSE. TALLY[n] and TOGS must be assigned a unitary-string or integer. The values apply at initialization only, since they are not protected (read only) fields within the station table.

Examples

```
INITIALIZE TOGS = 255.  
INITIALIZE TALLY[1] = 1, TALLY[2] = 4"7F",  
          TALLY[0] [4] = TRUE, TOG[2] = TRUE.
```

LOGIN

This statement specifies whether or not a station must log onto the system before communication can be continued. The station is logged in when the NDL receives a message header for the station with MCS data bit 14 set.

The format for the LOGIN statement is:

$$\underline{\text{LOGIN}} = \left\{ \begin{array}{l} \underline{\text{TRUE}} \\ \underline{\text{FALSE}} \end{array} \right\}$$

Example

```
LOGIN = TRUE.
```

MODEM

This statement defines the modem or data set to be used with this station. The modem-identifier must refer to a description previously defined in the Modem section.

The format for the MODEM statement is:

$$\underline{\text{MODEM}} = \text{modem-identifier.}$$

The modem on a station must be compatible with the modem on the line to which the station is connected. The selected modem must match the station in type and, if they are asynchronous, must have a speed greater than or equal to that of the station. If synchronous, they must be less than or equal to the station speed.

Example

```
MODEM = TA734.
```

All modems on a line must have the same speed.

MYUSE

This statement specifies whether a station is to be used for input only, output only, or both. The MYUSE statement is required.

$$\underline{\text{MYUSE}} = \left\{ \begin{array}{l} \underline{\text{INPUT}} \\ \underline{\text{OUTPUT}} \\ \underline{\text{INPUT, OUTPUT}} \\ \underline{\text{OUTPUT, INPUT}} \end{array} \right\}$$

A terminal REQUEST statement must be defined within the terminal description associated with this station for handling the capabilities specified in the MYUSE statement. Thus, if a station is to receive and transmit to the system, the terminal named in the station TERMINAL statement for that station must contain transmit and receive request identifiers.

Examples

MYUSE = INPUT.
MYUSE = INPUT, OUTPUT.

PAGE

This statement defines the number of logical lines per logical page. The value specified must be less than or equal to the number of lines specified for the terminal PAGE statement unless the number is zero, indicating that paging can be arbitrary.

The format for the PAGE statement is:

PAGE = integer.

If a PAGE statement is not included in the station description, the station's terminal specifications for paging are assumed.

Example

PAGE = 33.

Note

This data is available only to the MCS.

PHONE

This statement lists the telephone number associated with the station being described. The telephone number may not exceed 14 digits.

PHONE = integer.

Embedded hyphens are not allowed and should be deleted. The presence of the PHONE statement implies that the communications subsystem can dial the station, and that a modem has been specified as DIA-LOUT.

Example

PHONE = "2152698575".

Note

This data is available to the MCS only.

RETRY

This statement defines the number of retries that are to be accomplished for this station on a data communications error. For example, a transmission of message text to a station may be retransmitted up to the retry count specified. If the error continues, the communications subsystem notifies the system.

The format for the RETRY statement is:

RETRY = integer

The maximum value that may be expressed by the integer is 255.

Example

RETRY = 15.

SPEED

This statement is used to associate one of the speeds that is described in the terminal SPEED statement.

$$\underline{\text{SPEED}} = \text{integer} , . . . \left\{ \begin{array}{l} \text{(1)} \\ \text{(2)} \\ \text{empty} \end{array} \right\}$$

The integer must express the speed desired and the STOPBITS option must correspond to the terminal SPEED statement. This option is used for asynchronous terminals only. The SPEED statement is required.

Example

SPEED = 9600(1).

If a station is connected on an asynchronous line and a modem is on that line:

1. If neither the RATESELECT nor STANDBY options are specified in the modem's TYPE statement, only one speed may be specified.
2. If RATESELECT is specified, two speeds must be given with the higher value corresponding to the high value of the RATESELECT. The other speed is the low speed.
3. If both RATESELECT and STANDBY are specified, three speeds must be given. The highest one is the normal speed (i.e., STANDBY = FALSE). The other two speeds are the high and low speeds, and correspond to the RATESELECT setting when STANDBY is TRUE.
4. The station speeds must be less than or equal to the corresponding station's modem speeds and the line's modem speeds.
5. All stations must have the same number of stopbits.
6. A terminal must specify at least two speeds if it is on an asynchronous line having the RATESELECT options.
7. A terminal must specify at least three speeds if it is on an asynchronous line having both RATESELECT and STANDBY options.

If a station is on a synchronous line:

1. A synchronous station may specify only one speed.
2. The speed of the station must be greater than or equal to the fastest speed specified on that station's modem and line's modem.
3. A synchronous station's SPEED statement cannot specify stopbits.
4. The speed of a synchronous terminal must be greater than or equal to the fastest speed specified by that line's modem.
5. The speed of a terminal on a synchronous line must be greater than or equal to the fastest speed specified by that line's modem.

All stations on a line must specify the same speed(s).

SPO

If TRUE, this statement specifies that the station can be treated as a system SPO (console printer). This implies all system messages may be directed to the station and the station may input messages to the system as if it were a central system console printer.

The format for the SPO statement is:

$$\underline{\text{SPO}} = \left\{ \begin{array}{l} \underline{\text{TRUE}} \\ \underline{\text{FALSE}} \end{array} \right\}$$

Example

SPO = TRUE.

TALLIES

This statement specifies whether or not the additional station tallies (3-18) are available to the station.

The format for the TALLIES statement is:

$$\underline{\text{TALLIES}} = \left\{ \begin{array}{l} \underline{\text{TRUE}} \\ \underline{\text{FALSE}} \end{array} \right\}$$

If TALLIES is TRUE, there are 16 more tallies available for the station. The default value is FALSE.

The station's TALLIES statement must agree with the station's terminal TALLIES statement. (Refer to TALLIES statement in Section 6.)

Example

TALLIES = TRUE.
TALLIES = FALSE. % DOCUMENTATION ONLY

TERMINAL

This statement specifies the name of the terminal description that defines the physical characteristics of the station being described. The TERMINAL statement is required.

The format for the TERMINAL statement is:

$$\underline{\text{TERMINAL}} = \text{terminal-identifier.}$$

More than one station description may refer to the same terminal description.

Example

TERMINAL = TC500.

TYPE

This statement provides the selection of specific parameters that were specified in the terminal TYPE statement. If only one set of parameters was specified in the terminal TYPE statement, no further definition is required.

The format for the station TYPE statement is:

$$\underline{\text{TYPE}} = \left\{ \begin{array}{lll} \underline{\text{SYNC}} & \left\{ \begin{array}{l} \underline{\text{(MODEM)}} \\ \text{empty} \end{array} \right\} & \left\{ \begin{array}{l} \text{,BITS} \\ \text{empty} \end{array} \right\} \\ \underline{\text{ASYNC}} & \left\{ \begin{array}{l} \underline{\text{(TELEX)}} \\ \underline{\text{(DIRECT)}} \\ \underline{\text{(MODEM)}} \\ \underline{\text{(BDI)}} \\ \text{empty} \end{array} \right\} & \end{array} \right\}$$

If specified, the station TYPE must agree with the terminal TYPE. The BITS option may only be specified with SYNC and MODEM. If BITS is specified, it indicates that the station is capable of communicating in a bit-oriented line discipline environment.

Examples

TYPE = ASYNC.
TYPE = ASYNC (DIRECT).
TYPE = SYNC (MODEM), BITS

Note

If DIRECT, MODEM, BDI, or TELEX is not specified for TYPE = ASYNC, only one ASYNC type may be specified in the terminal TYPE statement.

WIDTH

This statement specifies the maximum number of characters in a logical line of output. The value defined must be less than or equal to the number of characters specified in the WIDTH statement of the associated terminal description.

The format for the WIDTH statement is:

WIDTH = integer.

If no station WIDTH statement is included in the station description, the information specified for the terminal TYPE associated with the station is used as a default width size.

Example

WIDTH = 72.

Note

This data is available only to the MCS.

WRAPAROUND

If TRUE, this statement indicates that the station's terminal prevents the truncation of output messages whose length is greater than that specified by the WIDTH statement.

The format for the WRAPAROUND statement is:

WRAPAROUND = $\left\{ \begin{array}{l} \underline{\text{TRUE}} \\ \underline{\text{FALSE}} \end{array} \right\}$.

If no station WRAPAROUND is specified, the terminal WRAPAROUND specification is used.

Example

WRAPAROUND = TRUE.

Note

This data is available only to the MCS.

SECTION 8

LINE SECTION

GENERAL

This section provides the logical and physical characteristics of the lines to the communications subsystem. For each line on the data communications system, this section provides the initial state of the lines and their physical relationship to the stations in the network. A line definition is needed for all lines on the system.

LINE DEFINITION DESCRIPTION

For each line on data communications network, a line definition must exist. Lines with like characteristics may use the line DEFAULT statement to reduce source code. The line TYPE must be explicitly stated.

The general format for a line definition is:

$$\underline{\text{LINE}} \quad \left\{ \begin{array}{l} \text{DEFAULT} \\ \text{empty} \end{array} \right\} \quad \text{line-identifier:} \quad \left\{ \begin{array}{l} \text{line-attribute-statement-1.} \\ \text{line-attribute-statement-2.} \\ \cdot \quad \cdot \quad \cdot \\ \cdot \quad \cdot \quad \cdot \\ \text{line-attribute-statement-n.} \end{array} \right\} \quad \dots$$

If DEFAULT is specified, the compiler recognizes that the line-attribute-statements specified are to be associated with the specified line-identifier. When the line-attribute-statement DEFAULT is specified, the compiler uses the line-identifier of the DEFAULT statement to decide which line DEFAULT definition is to be associated with that line. A DEFAULT line definition is not considered to be attached to the system.

Example

```
LINE DEFAULT LEASEDLINE:  
  MODEM = TA783.
```

The line DEFAULT statement further explains this format. All statements end with a period.

ADDRESS

This statement is used to specify the physical address of the described line. This statement is required.

The format for the ADDRESS statement is:

$$\begin{aligned} \underline{\text{ADDRESS}} &= \text{addr1} \quad \text{addr2} \\ \text{addr1} &= \left\{ \begin{array}{l} \text{integer} \\ \text{empty} \end{array} \right\} \\ \text{addr2} &= \left\{ \begin{array}{l} \text{:integer} \\ \text{empty} \end{array} \right\} \end{aligned}$$

The ADDRESS statement is different for each of the CMS systems.

B 80 Systems

1. <addr1> is the channel number (0-7).
2. <addr2> is the sub-channel number (0-2) or empty.

B 700/B 800 Systems

1. <addr1> is the PMLC number (0-1)
2. <addr2> is the line number (0-15).

B 1800 Systems

1. <addr1> is the logical DCP number (0-4).
2. <addr2> is the adapter number (0-6).

CP9550 Systems

1. <addr1> is the logical DCP number (0-7).
2. <addr2> is the line number (1-7).

Example

ADDRESS = 0:2.

DEFAULT

This statement is used to specify the name of a set of line-attribute-statements to be associated with a line. For lines with like characteristics, it is advantageous to group these characteristics under a DEFAULT definition and list any additional statements under each individual line definition. The NDL compiler then refers to the default list to complete the line specification.

The format for the DEFAULT statement is:

$$\underline{\text{DEFAULT}} = \text{line-default-identifier.}$$

The line-default-identifier must have been described previously in a line DEFAULT definition. If the DEFAULT statement is used, it must be the first statement in the line definition.

Example

```
LINE TC01:  
  DEFAULT = LEASEDLINE.  
  STATION = TC51A.
```

MAXSTATIONS

This statement specifies the maximum number of stations that may be attached to a line. The MAXSTATIONS statement may be specified for any line that has stations or has the capability of having stations.

The format for the MAXSTATIONS statement is:

MAXSTATIONS = integer.

The maximum number of stations per line is the smaller of 99 of the number of stations specified in the Station section. If the MAXSTATIONS statements are not specified, it is assumed that the maximum number of stations equals the number of stations that are explicitly specified as being on the line. If no stations are assigned to the line, MAXSTATIONS must be greater than 0.

Example

```
MAXSTATIONS = 86.
```

MODEM

This statement defines the modem that is to be associated with this line. The modem-identifier must refer to a description previously defined in the Modem section.

The format for the MODEM statement is:

MODEM = modem-identifier

The line MODEM must be compatible with the modems on each station on the line.

Example

```
MODEM = SUPERMODEM.
```

RATESELECT

Some modems have the ability to operate at two different speeds. If the modem used on the line has this ability, this statement specifies the position at which the modem is to be set initially.

RATESELECT = $\left\{ \begin{array}{c} \text{HIGH} \\ \text{LOW} \end{array} \right\}$

This statement must occur if line MODEM specifies RATESELECT.

STANDBY

This statement specifies whether, at system initialization, the modem is operating at its normal speed (STANDBY or FALSE) or one of its standby speeds.

$$\underline{\text{STANDBY}} = \left\{ \begin{array}{l} \underline{\text{TRUE}} \\ \underline{\text{FALSE}} \end{array} \right\}$$

If this statement is not specified, STANDBY is false.

STATION

This statement identifies the stations associated with that line. The station-identifiers used in the Station section are acceptable entries in this statement.

The format for the STATION statement is:

$$\underline{\text{STATION}} = \text{station-identifier-1, ... station-identifier-n.}$$

Example

STATION = TC51A, TC51B, TC5AA, TC7BB.

The number of stations specified may not exceed MAXSTATIONS.

TYPE

This statement specifies the state of the associated line, whether it is a DIALIN or DIALOUT line.

The format for the TYPE statement is:

$$\underline{\text{TYPE}} = \left\{ \begin{array}{l} \underline{\text{DIRECT}} \\ \underline{\text{MODEM}} \\ \underline{\text{BDI}} \\ \underline{\text{TELEX}} \end{array} \right\} \left\{ \begin{array}{l} \text{empty} \\ , \underline{\text{DIALIN}} \\ , \underline{\text{DIALOUT}} \\ , \underline{\text{DUPLEX}} \\ , \underline{\text{BITS}} \end{array} \right\} \left(\left\{ \begin{array}{l} \underline{\text{ACU}} \\ \underline{\text{MODEM}} \end{array} \right\} \right) \dots \left. \right\}$$

DIRECT

This indicates that there is no data set interface and that the connection is a two-wire direct connect type.

BDI

This indicates a balanced differential interface line. This is a type of three-wire direct-connect; therefore, no modem should be specified.

MODEM

If not further qualified, this option implies that the line is a leased line.

TELEX

A line which may be dialed from a remote station or a group of remote stations is a DIALIN line.

A line that has telex units attached to it. All stations on the line must have TELEX specified as their type.

If TELEX is specified:

1. STANDBY is not allowed.
2. If terminal or station specifications used on that line allows TELEX, then RATESELECT is not allowed.
3. The terminal and station type specifications must be asynchronous.

If no TYPE is specified, the default is an unqualified MODEM or DIRECT, depending upon the presence of a MODEM statement.

DIALIN

A line which may be dialed from a remote station or a group of remote stations is a DIALIN line.

DIALOUT

A line that is attached to an automatic calling unit (ACU) and is capable of dialing out to a data set is called DIALOUT. The DIALOUT line and its associated ACU may be used by the system to call up any compatible remote station which can be dialed.

DUPLEX

A line that transmits and receives on two different physical wires without requiring line turnaround times is called a DUPLEX line.

BITS

A line that is capable of communicating in a bit-oriented line discipline environment is called BITS (i.e., BDLC line discipline).

Examples

TYPE = DIRECT.
TYPE = MODEM, DIALIN.
TYPE = MODEM.
TYPE = MODEM, DIALOUT (ACU).
TYPE = MODEM, DIALOUT (MODEM), DUPLEX.
TYPE = MODEM, DUPLEX, BITS.

SECTION 9

DCP SECTION

GENERAL

This section provides the NDL programmer with the capability to describe the main memory buffer configuration which best suits the environment. If the DCP section is omitted, default values are calculated from other network parameters.

DCP Definition Description

The format for the DCP definition is:

DCP identifier:

<u>LIMIT</u>	= integer. *
<u>BUFFERCOUNT</u>	= integer. *
<u>BUFFERSIZE</u>	= integer. *
<u>CODEFILE</u>	= identifier.
empty	
<u>MEMORY</u>	= integer.
<u>TERMINAL</u> [phrase]	= identifier.
DCP [<identifier>] MEMORY	= integer.

*These statements are not required, but if any one is specified, then all three are required.

There may only be one DCP section in the NDL program.

BUFFERCOUNT

This statement allows the NDL programmer to specify the minimum buffer count for the system. This statement is required.

BUFFERCOUNT = integer

The integer specified must be greater than 3. Caution should be used when specifying large buffer counts and large buffer sizes that excessive amount of memory are not being used.

Example

BUFFERCOUNT = 10.

BUFFER

This statement allows the NDL programmer to specify the size of the data communications buffer in bytes. This statement is required.

BUFFER = integer.

The integer specified must be greater than 41. Caution should be used when specifying large buffer counts and buffer sizes that excessive amounts of memory are not being used.

Example

BUFFER = 256.

CODEFILE

This statement allows the NDL programmer to specify a name for the NDL object code file; this statement is optional. If it is not specified, the default code file name is NDLSYS.

The format for the CODEFILE statement is:

CODEFILE = "identifier".

The identifier is less than or equal to 12 characters.

Example

CODEFILE = "COD776".

DCP MEMORY

This statement specifies the amount of memory on a particular DCP on a system.

The format of the DCP MEMORY statement is:

DCP [identifier] MEMORY = integer

In the DCP MEMORY statement, the identifier tells which DCP is being referenced. Identifiers can have a value from 0 to 7.

Integer specifies the amount of memory on a particular DCP. It must have one of the following values: 6144, 8192, 32768, 49152, or 65536. The default is 65535.

Example

DCP [2] MEMORY = 8192.

MEMORY

This statement allows the NDL programmer to specify the amount of main memory in the system in bytes. This statement is documentary only.

The format for the MEMORY statement is:

MEMORY = integer.

Example

MEMORY = 98304.

LIMIT

This statement allows the NDL programmer to specify the maximum number of buffers that will be allocated by the system. This statement is required.

LIMIT = integer.

The integer must be equal to or greater than the value for BUFFERCOUNT. The difference between LIMIT and BUFFERCOUNT determines the size of the reserve buffer pool.

If there is no DCP section, the total buffer space will not exceed 14,000 bytes.

Example

LIMIT = 20.

TERMINAL

This statement is used by the NDL programmer to provide information for the post processor program (refer to Section 12) which creates the non-interpretive NDL.

TERMINAL ["identifier" (0 to 7) (LOAD
empty)] =
(terminal-id, . . .)
ALL

The identifier within quotes (") is the name that the post processor program assigns to the non-interpretive DCP code file. The "identifier" must be 12 or fewer characters. The number following the identifier specifies to which DCP the non-interpretive code file is to be loaded. LOAD specifies that the program-id specified in this statement is the code file loaded to the specified DCP at MCS execution time. If LOAD is not specified, NDLDCP is loaded at MCS execution time and used with all defined terminals, if the corresponding DCP memory size is 6144. Otherwise, the LOAD file is BDLDCP with all terminals.

The terminal-ids are the identifiers specified in the Terminal section of NDL. These identifiers are used by the post processor to identify which request sets are to be put into the non-interpretive NDL. If ALL is specified, all the Terminal section's request sets are included in the non-interpretive NDL. A maximum of five TERMINAL statements may be specified per DCP.

The "identifier" may be NDLDCP, in which case, the post processor program ignores the information in this statement, but the REDEFINE.STATION statement uses it to determine whether or not the terminal

may be used.

Examples

```
TERMINAL ["ABC" (3) : LOAD] = TD 800, TD 700,  
TERMINAL ["BRDBAUD" (1) : LOAD] = ALL.  
TERMINAL ["NDLTEST1" (0)] = TD 800, TD 700, TC500.
```

SECTION 10

FILE SECTION

GENERAL

This section of a network definition associates stations that are part of the network with the data comm file of an object job(s). The stations referenced in the File section must be defined in the Station section of the network definition.

File Definition Description

The general format for a FILE definition is:

FILE file-identifier: family-statement

The file-identifier assigned by the NDL programming is associated with the family description following it.

Example

FILE BANKONE:

FAMILY = STATION1.

FAMILY

This statement defines the station identifiers, or other files that are to be considered members of that family and associated with that file-identifier.

The format for the FAMILY statement is:

$$\underline{\text{FAMILY}} = \left\{ \begin{array}{l} \text{station-identifier} \\ \text{file-identifier} \end{array} \right\} \dots$$

If the file-identifier option is used as part of the family description, it must have been previously defined in the File Section.

The station-identifier option is used to specify all the stations associated with that file-identifier. All stations must have been previously defined in the Station section.

Examples

FILE BANKONE: FAMILY = TC51A, TC7BA, TU7RR.

FILE BANKTWO: FAMILY = TC701, TC702, TC703.

FILE ALLBANKS: FAMILY = BANKONE, BANKTWO, TC3CA.

Note

All file identifiers must be unique in the first 12 characters.

SECTION 11

OPERATION OF CMS NDL COMPILER

GENERAL

This section describes the on-board CMS NDL compiler and sets forth requirements for its use. It is intended for use on Burroughs CMS Data Communications Subsystems. The compiler is capable of accepting input from 80-column cards, magnetic tape, tape cassette, system operator's console, or disk. The input to the compiler is in 80-column records with characters 1 through 72 being statements in free form and characters 73 through 80 being a sequence number. Output listings are available on a 132-column line printer or on the system operator's console. The resulting object file is placed on disk.

HARDWARE

The minimum system configuration required for use with this compiler is as follows:

1. Processor - Any CMS processor.
2. Main Memory - 24 K bytes (plus operating system requirements).
3. Disk - 3000 segments.
4. System Operator's Console - Compiler execution must be initiated on the system operator's console.

Optional devices which can be used with this compiler are as follows:

1. Line Printer.
2. Magnetic Tape.
3. Tape Cassette.
4. Card Reader (80 or 96-column).

SOFTWARE

Any CMS software release.

COMPILER OPERATION

The CMS NDL compiler is an on-board compiler designed to run on any CMS system. Some of the optional capabilities may not be available or may apply to certain CMS systems.

Execute

Compiler execution must be initiated from the system operator's console using the following format:

```
[EX] CMSNDL1 [ { value clause  
file equate clause  
SYNTAX } ] ...
```

Note

Brackets indicate optional information. The syntax is explained below.

Value Clause

The value clause gives the user the ability to specify \$-Options in the EXECUTE statement instead of in a card deck, making it possible for two non-card sources to be used as the master and patch files. Only one value clause is allowed in an EXECUTE statement. The format is:

VA 0 [=] ABCDEF 680020

Each bit in the integer ABCDEF represents a compiler option as follows:

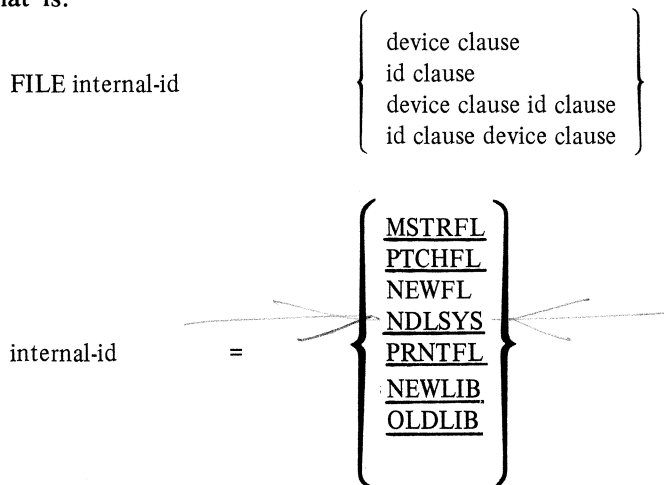
Section 11
 Operation of CMS NDL Compiler

Digit	Bit	Description
A	8	Patch symbolic input is on disk
	4	Patch symbolic input is on disk (same as 8)
	2	Patch symbolic input is on tape
	1	Patch symbolic input is on cards (default)
B	8	DISK
	4	DISK (same as 8)
	2	TAPE
	1	CARD (master symbolic input is on cards)
C	8	NEWD
	4	NEWD (same as 8)
	2	NEWT
	1	NEWC
D	8	MTCH
	4	BLNK
	2	LGEN
	1	LIBR
E	8	CODE
	4	LIST
	2	LST1
	1	SUPR
F	8	RESERVED
	4	RESERVED
	2	NPRT
	1	TBLS

Definitions of these options are given later in this section.

File Equate Clause

The file equate clause gives the user the ability to specify different names for input and output files and to clarify to which type of device a \$-Option refers (such as 80 or 96-column card reader). There may be up to eight file equate clauses in an EXECUTE statement and the device types must agree with the \$-Options specified. The format is:



device clause = $\frac{\text{DEV}}{\text{DEVICE}}$ [=] device

device = $\left\{ \begin{array}{c} \text{CARD} \\ \text{CR96} \\ \text{CONS} \\ \text{TAPE} \\ \text{CASS} \\ \text{DISK} \\ \text{LP} \end{array} \right\}$

id-clause = $\frac{\text{ID}}{\text{ID}}$ [=] $\left\{ \begin{array}{l} \text{file-id} \\ \text{pack-id/file-id} \end{array} \right\}$

The following is a list of all compiler user files along with their associated \$-Options, where applicable.

Name	Description	\$-Option
MSTRFL	Master Symbolic Input	DISK, TAPE, CARD
PTCHFL	Patch Symbolic Input	(See Note)
NEWFL	New Symbolic File	NEWC. NEWD. NEWT
NDLSYS	NDL Object File	
PRNTFL	Source Listing	LIST, LST1, TBLS
NEWLIB	New NDL Library	LGEN
CLDLIB	Old NDL Library	LIBR

Note

When patch input is from cards, the expected file-id is CARD.

Each file equate may have a device specification, an id-specification, or both. The device clause determines which device the file is on. The id-clause specifies the file and pack name, if it is other than the default name.

File equate information may also be entered through a card patch file. The file equate card(s) must precede all other cards in the deck and have a “#” in column 1.

File equate cards are not included in the generation of new source files.

Syntax

Specification of the SYNTAX option prevents all code and table generation, thereby reducing compiler times during program development.

Continuation

The EXECUTE statement is limited to 72 characters; if greater than 72 characters, a hyphen may be entered in place of the 72nd character. The compiler will then accept further information via the "AX" facility.

Console Input

Patch input may be entered through the system operator's console by file equating the PTCHFL to the console. When operating in this mode, the compiler first accepts all leading \$-Options. When the first card without a "\$" in column 1 is encountered, the compiler prints a message indicating the leading options have been completed. From this point on, the first eight columns must contain a sequence number, with the actual text in columns 9 through 80. The compiler reverses the fields, creating a normal card image.

Source Merge Process

When two separate source files are to be combined for input, one file is designated as the master and the other as the patch. If there is only one source input file, it is considered the patch.

The merge pattern is controlled by sequence numbers in columns 73 through 80 (1 through 8 for SPO input). Patch images are inserted into the source file at those locations determined by the patch sequence number. If a patch and master source image have the same sequence number, the patch takes precedence. Any source images without sequence numbers immediately follows the previous source image with a sequence number. Blank sequence numbers are disallowed for console input.

Sequence numbers in both the patch and master files must be in ascending order. Any source image occurring out of order results in a sequence error warning.

Dollar (\$) option cards may only appear in a patch file.

Library Generation and Usage

NDL libraries containing precompiled NDL control and request sets can be generated and used by the compiler. This eliminates the need for syntaxing and compiling (which are often used) and reduces compile time.

To create a library, the LGEN \$-Option must be specified and the NDL program may contain only Constant, Control, and Request sections. The generated library is a single disk file called NEWLIB.

To reference library procedures, the LIBR \$-Option must be specified. The compiler expects an NDL library (OLDLIB) to be resident on disk. To invoke any single procedure, the following syntax is used:

CONTROL LIBRARY <library control procedure name>:

REQUEST LIBRARY <library request procedure name>:

The procedure names are those that were used when creating the library. If any statement other than a new section header follows a library procedure call, a syntax error occurs.

Whenever the LIBR \$-Option is specified, all constants defined when creating the library are included in the referencing program. Therefore, care must be taken not to declare new constants with the same names.

COMPILER OPTIONS

Option Format

All option cards are identified by a dollar sign (\$) in column 1. Options are specified in columns 2 through 72 and may be separated by commas (,).

Option Control

Individual options may be set or reset according to the following rules:

1. The occurrence of the word SET causes options following it to be set.
2. The occurrence of the word RESET causes all options following it to be reset.
3. If SET or RESET is not specified, all options are reset and the options specified on the card are set.
4. When SET and RESET are used, they must be the first word following the "\$". Only one may occur per card.
5. All options specified prior to the first non-option card (leading options) have a SET condition implied.
6. Certain options can only occur as leading options and cannot be reset. These are:

CARD	NEWC	LGEN
DISK	NEWD	LIBR
TAPE	NEWT	TBLS

DOLLAR OPTIONS

LIST

Generates a full double space listing.

LST1

Generates a full single space listing.

VOID NNNNNNNN

Removes all source images from the point of insertion up to and including the source image with sequence number NNNNNNNN.

RSEQ NNNNNNNN +MMMMMMMM

Causes all card images to be given new sequence numbers starting at NNNNNNNN and incrementing by MMMMMMMM for each successive card. Option cards are not included. Default values are 0000100 and +00001000.

CODE

Causes the code generated for control and request procedures to be output in the listing.

MTCH

Matches each BEGIN with its corresponding END on the listing.

SUPR

Suppresses warning messages from the listing.

NPRT

Suppresses the listing of the summary at the end of the compilation.

SPEC

Negates LIST and LST1 if any errors are detected.

LNXX

Specifies that XX lines are to be printed per page. XX must be greater than 4.

CARD

Indicates that master input is from cards.

DISK or DISC

The master symbolic input file resides on disk.

TAPE

The master symbolic input file resides on tape.

NEWC

Requests that a new card deck be punched.

NEWD

Requests that a new symbolic disk file be created.

NEWT

Requests that a new symbolic tape file be created.

LGEN

Causes a new NDL library to be generated.

LIBR

Specifies that the program references an old NDL library.

TBLS

Requests that the generated NDL tables be printed.

If no options are specified, patch input is from cards and no listing is printed.

COMMENTS

The percent sign (%) is used to terminate a card image. All information to the right of a “%” is treated as comments.

TOP OF PAGE

The top-of-form can be generated by specifying “@PAGE” in columns 1 through 5. The rest of the card is ignored.

SECTION 12

B 800 POST PROCESSOR PROGRAM

GENERAL

The capability of enhanced character throughput on the DCP is provided through the implementation of a post processor program (PPP). Enhanced throughput is achieved by emitting microcode from the post processor, thus eliminating the need to fetch S-OPS. The post processor is tuned for the specific configuration(s) defined by the NDL programmer. The PPP uses the NDLSYS file (output of the NDL compiler) as input in the generation of microcode.

When executed, the PPP interrogates the NDLSYS file for information directing the PPP to generate a microcode file for DCP. This information is supplied in the DCP terminal statement through NDL. From this and other information within the NDLSYS, the PPP program generates a code file identified by the name supplied in the DCP terminal statement. The data comm loader recognizes this identifier at MCS start-up time and loads the proper microcode file into the DCP.

POST PROCESSOR ACTION

The data segment in the NDLSYS containing the information generated by the DCP terminal statement is used as follows:

1. A search is conducted for the file names which are not NDLDPC. If there are no such files, the program terminates.
2. Upon encountering a non-NDLDPC file, the PPP accesses the terminal list for that file and the DCP number of the DCP on which the firmware is executed.
3. The terminal list is scanned to determine whether the particular file is multiline or single line. The decision is made as follows:
 - a. If all terminal names are assigned through stations to lines, and
 - b. Those assignments are all to one line, then single-line is assumed.

Effectively, if a terminal(s) does exist on one and only one line per DCP, the file is marked as a single line case (i.e., a higher throughput rate is provided though the single-line case permits only one active line at any one time). Other lines can be used providing a valid terminal is assigned. Using different lines, nonsimultaneously, can be accomplished by making the line that was previously used not ready, reconfiguring the particular terminal type(s) to another line, and making it ready.

Execution of Post Processor Program

The post processor requires no inputs other than the NDLSYS file, and can be initiated by the EX PPP command.

At the completion of the PPP, a microcode file exists for each file identifier named within the NDLSYS.

Data Comm Loader

The data comm loader is responsible for loading the generated firmware file to the DCP(s). At MCS start-up time, the loader interrogates the NDLSYS file for the file name of the firmware that is to be loaded. The firmware file specified through the DCP terminal LOAD statement is the initial file loaded to the DCP.

Microcode Listing

During the code generation of the PPP, a disk file of microsource images is created. After code generation of a particular file is completed, the disk file is listed to the line printer. The listing contains information for each instruction generated. Any unused microspace is deleted from the listing. The following information is displayed:

1. S-instruction mnemonic, if pertinent. If the microcode being generated is a direct result of a particular S-instruction, that S-instruction is printed.

Section 12

Non-Interpretive NDL

2. Hexadecimal S-level address.
3. Hexadecimal contents of the S-instruction.
4. Any microcode label. If the instruction being printed is a reference label, that label is printed.
5. Symbolic representation of the microinstruction.
6. Hexadecimal address of the microinstruction.
7. Hexadecimal value of the microinstruction.

SECTION 13

NDL POST COMPILER

INTRODUCTION

This section describes the operating procedures and functions of the CP 9550 NDL Post Compiler (NPC) for those readers involved in the implementation of data communications on the CP9500 series. The information in this section is divided into two parts. The first part provides an overview of NPC with respect to operating instructions, error messages, and requirements for execution. In order to understand this, the reader should be familiar with CP 9550 operations and the NDL compilation process.

The second part gives a detailed description of NPC program flow. The reader should have knowledge of the CMS NDL code file structure, CMS data communications, and CMS NDL source language. Knowledge of the CMS NDL S-machine and CP 9550 microcode is useful, but not essential.

Related Publications

CMS Data Communications Reference Manual, form 1090909.

OVERVIEW

CMS NDL source is compiled to produce a program file; the structure of which is common for all CMS systems. This program file consists of tables containing: 1) data, which defines the characteristics of the communications network; and 2) code, which performs line and station-oriented functions. The code contained within the program file is designed to be interpreted. That is, each operation defines an action to be taken by an interpreter. The manner in which this code is treated is dependent on the individual CMS implementation. It may either be left in the form produced by the NDL compiler or, in order to eliminate the need for operation fetching and decoding, it may be translated into the microcode of the individual CMS machine. The latter approach is taken on the CP 9550 series. That is, after an NDL program file is produced, this is translated into a form consistent with the CP 9550 implementation. The program which performs this translation is NPC, an MPL program, which may be run on any CMS machine.

OPERATING PROCEDURES

Before executing NPC, the NDL program file must be produced via an NDL compilation (see figure 13-1).

The DCP TERMINAL statement (Section 9) directs the operation of NPC. The following paragraphs describe the use of the DCP TERMINAL statement with respect to the CP9500 implementation.

The DCP TERMINAL statement specifies:

1. The valid terminal(s) for the DCP files produced.
2. The name of each DCP file.

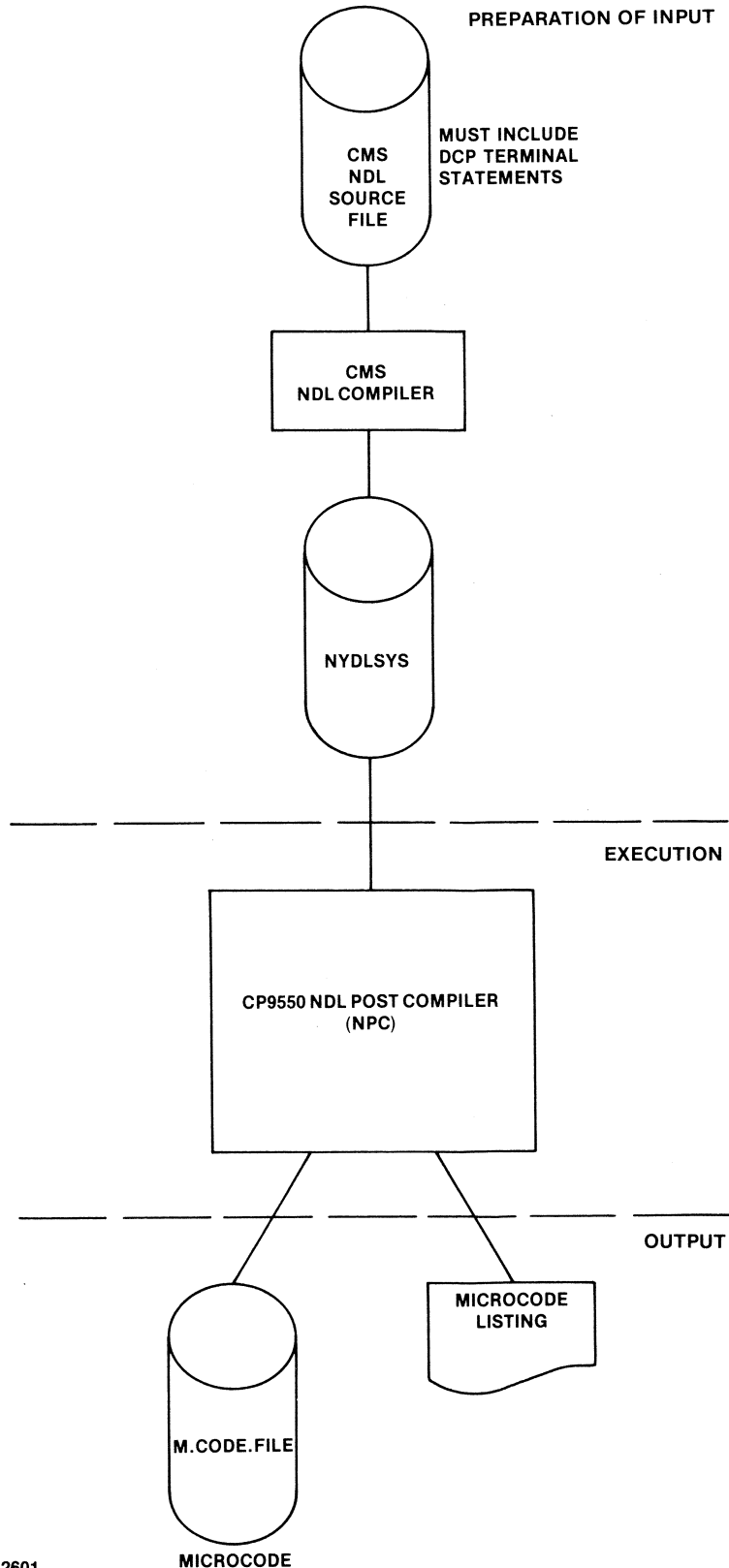
Because the firmware file is tuned for a particular configuration, any terminal types not named are invalid for that file.

The DCP TERMINAL statement syntax serves a dual role:

1. Identifies the DCP and its associated terminal types for each NPC-generated file.
2. Specifies loading an NPC-generated file at MCS start-up time.

The format of the DCP TERMINAL statement is as follows:

$$\langle \text{DCP TERMINAL STMT} \rangle :: = \text{TERMINAL} [\langle \text{PROGRAM NAME} \rangle \langle \text{DCP \#} \rangle \langle \text{LOAD OPTION} \rangle] = \langle \text{TERMINAL NAME LIST} \rangle$$



ED2601

MICROCODE

Figure 13-1. NDL Post Compiler Flow

Where:

```
<program name> ::= <string>
<DCP #> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
<load option> ::= : LOAD | <empty>
<terminal name list> ::= <terminal list> | ALL
<terminal list> ::= <terminal id> | <terminal list> , <terminal id>
```

In addition to the NDL source restrictions (Section 9), NPC imposes the following restrictions:

1. At least one of the five allowable DCP TERMINAL statements for each DCP must contain a LOAD clause.
2. The <terminal name list> may specify a maximum of 16 terminal types.
3. Because line address 0 is reserved for processor interface use, the maximum number of lines per DCP is seven; that is, the valid line addresses are 1 through 7.
4. One DCP is allowed per CP 9550 system.

Examples

```
TERMINAL [BISYNC (0)] = BSCTERM.
```

This statement causes a microprogram named BISYNC to be generated for DCP 0. The program is tuned to use the terminal BSCTERM.

```
TERMINAL [TD8 (0)] = TD820, TD800.
TERMINAL [HOST (0): LOAD] = B 4700.
```

These statements cause two microprograms to be generated. Both are generated for use on DCP 0. The programs are named TD8 and HOST. Program TD8 is capable of running with terminals TD820 and TD800. Program HOST is capable of running with terminal B 4700. AT MCS start-time, the program HOST is loaded to DCP 0.

EXECUTING NPC

NPC allows the user to select run-time options in the initiating message. The following is the syntax for the initiating message:

```
<NPC execute string> ::= EX NPC900 <del> <initiating message>
<initiating message> ::= <initiating message item> |
    <initiating message> <del> <initiating message item> | <null>
<del> ::= <blank>
<initiating message item> ::= DBG |
    NO.LIST |
    FILE <pack-id> /<file.id> |
    WORK <pack.id> |
    STATUS
```

The following paragraphs describe the <initiating message item> option list.

DBG

As the name implies, this is the DEBUG option used for compiler debugging. It causes pertinent DCP run-time information to be dumped to printer. The default for this option is the suppression of the debug listing.

NO.LIST

This option provides for the suppression of the microcode listing for each of the code files generated. The default for this option is the production of microcode listings.

FILE

This is the file equate statement. It allows the user to specify the PACKID/FILEID of the input file. The <pack.id> and <file.id> must conform to CMS definitions.

WORK

This option forces NPC work files to specified non-system disks. The default for this option is that the work files reside on system disk.

STATUS

This option causes the NPC to send display messages to the SAO at various times during execution. This option informs the operator that the program is not looping or otherwise hung. It is particularly useful when compiling large files with numerous terminal types. The default for this option suppresses all status messages.

NOTE

If the FILE option is used, all generated microcode is on the specified <pack.id>.

Any deviation from the defined syntax causes the termination of NPC.

Examples

1. EX NPC900 FILE NDLCODE.
 - a. NPC looks for a file called NDLCODE on system disk.
 - b. A listing is produced.
 - c. All generated DCP files are on system disk.
2. EX NPC900 NO.LIST FILE USER/NDLCODE.
 - a. NPC looks for a file called NDLCODE on pack user.
 - b. No listing is produced.
 - c. All generated DCP files are on pack USER.

ERROR MESSAGES

NPC emits error messages when any one of the following conditions occur:

Firmware file exceeds memory size.	Generation of the current DCP file has terminated because NPC has detected that it requires more memory than is available in the DCP (64 KB).
Invalid NDL S.OP is encountered.	An Op-code located in the NDL program is not implemented on the CP 9550, or the NDL program file has been corrupted.
Unable to resolve S-level branch address.	The NDL program file has been corrupted, or a bug exists in either NPC or the NDL compiler.
Invalid line number detected.	The NDL source line section specifies air address outside the range of 1-7.
Unresolved microaddress (NPC bug). Subroutine nesting error (NPC bug). File contains more than 16 terminal types.	

Definitions

Before proceeding, the reader should be familiar with the following terms and their definitions.

NDLSYS

The generic term used to describe the program file generated by the NDL compiler.

S-CODE

Secondary code produced by the NDL compiler. Consists of S-Ops and operands.

S-OP

Operation code defining an action to be performed on the associated operands by the interpreter.

S-LABEL/S-ADDRESS

A location within the S-code.

LABEL RESOLUTION

The process by which the reference to S-labels of unknown microaddress are supplied with the correct microaddress once that address has been determined.

MANAGER SCHEME

By using this method, the generated microprogram shares the processor between lines belonging to that processor.

HOST CONTROL

The interface between the operating systems and the DCP. By using this method, the operating system transfers requests (which require DCP action) to the DCP.

POST COMPILER/NPC MEMORY

The data areas (local to NPC) are used during code generation to hold table information read from the NDLSYS file.

NPC Functions

NPC uses the NDLSYS file (as produced by the NDL compiler) to perform the following tasks:

1. Translate NDL S-code into microcode using information contained in the NDL tables.
2. Generate a set of microcode corresponding to the terminal line control/request sets for each unique NDL terminal type.
3. Optimize microcode based on characteristics of NDL terminal types.
4. Build a microcode file for each DCP TERMINAL statement specified in NDLSYS.

Figure 13-2 illustrates NPC program flow.

In addition to the NDL-related functions described previously, the microcode file generated by NPC must also perform the following general functions:

1. Buffer management.
2. Message/buffer linking.
3. Table indexing in DCP memory.
4. Translation table space allocation and filling.
5. Processor register use.

Program Structure

NPC contains three logical divisions:

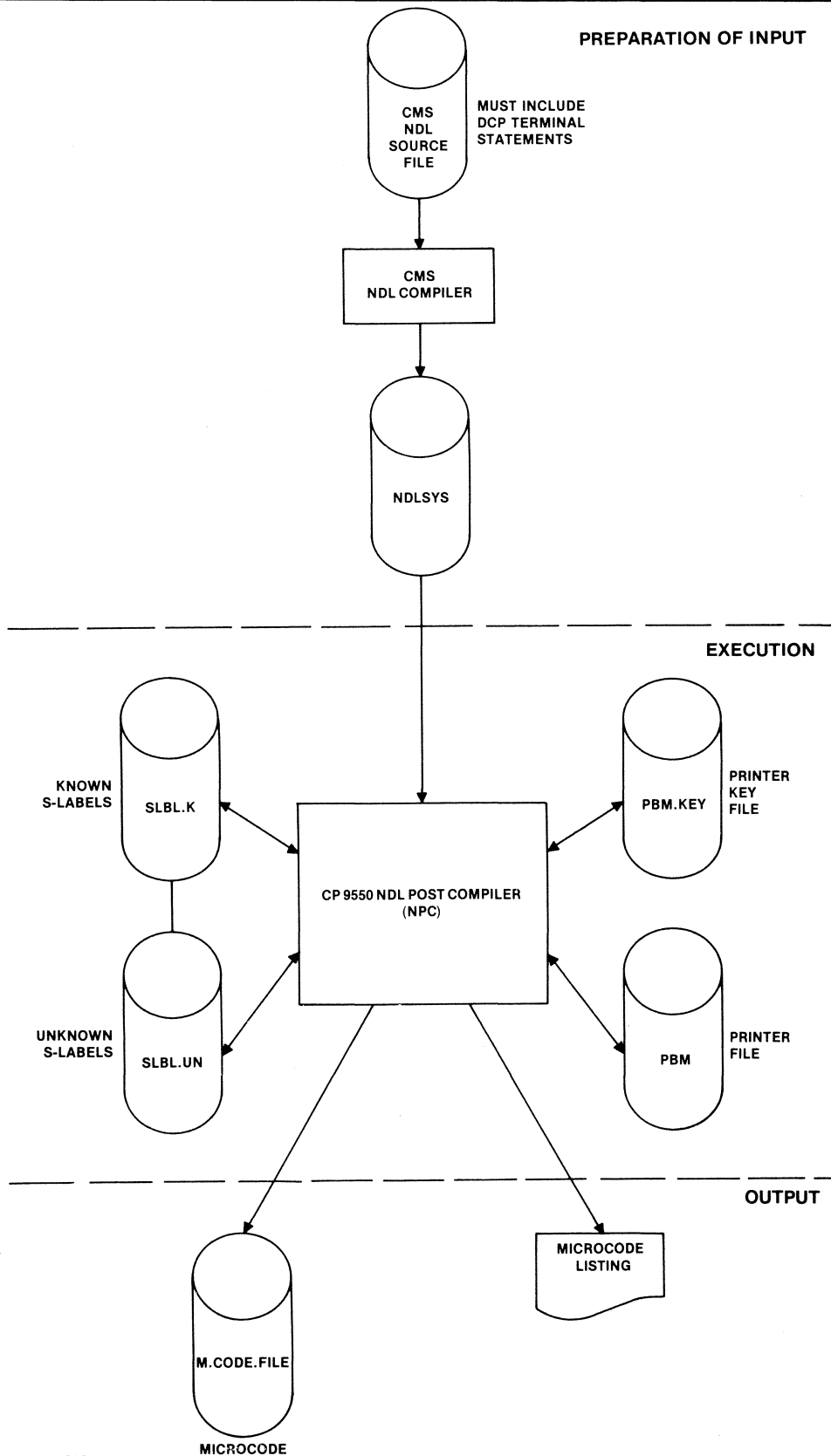
1. Environment establishment.
2. Microcode generation.
3. Microcode listing generation.

Succeeding paragraphs briefly describe each phase of NPC execution.

Environment Establishment

The establishment process organizes the initial data required for subsequent post processing. It performs the four following major functions:

1. Retrieves information from the NDLSYS file, that is, output file names from the DCP TERMINAL statement and the terminal list for each file specified.
 2. Places the retrieved data in post-processing memory.
 3. Allocates data memory.
-



ED2602

Figure 13-2. NDL Post Compiler Flow

4. Opens and readies those disk files used during the microcode generation process.

Microcode Generation

The establishment process and the microcode generation process are integrally related to each other. After output files, workfiles, and tables are established, the microcode generation process analyzes object code in NDLSYS and generates microcode.

Following is a list of major microcode generation functions:

1. S-instruction processing.
2. S-label generation.
3. Label resolution.
4. Line discipline generation.
5. Subroutine generation.
6. Microcode and data storage.
7. Manager schemes.
8. Code Optimization.

Microcode Listing Generation

The microcode generation process creates a disk file of microsource images. When code generation is complete, NPC automatically lists the file's contents on the line printer. This listing contains information for each instruction generated, as follows:

1. The S-instruction mnemonic, if pertinent. If the microcode being generated is a direct result of a particular S-instruction, that S-instruction is printed.
2. Hexadecimal S-level address.
3. Hexadecimal contents of the S-instruction.
4. Any microcode label. If the microinstruction being printed is a referenced label, that label is printed.
5. Symbolic representation of the microinstructions.
6. Literals or addresses referenced.
7. Comment field (if supplied).
8. Hexadecimal address of the microinstruction.
9. Hexadecimal value of the microinstruction.

File Structure

NPC uses disk files as described in succeeding paragraphs.

Generated Microcode File (M.CODE.FILE)

This is the output file containing the generated microcode. Its internal attributes are as follows:

INTERNAL FILE.NAME:	M.CODE.FILE
MFID:	0000000
NO.BUFFERS:	2
RECORD:	180 bytes
CLOSEMODE:	Lock
FILETYPE:	@17@
ACCESSMODE:	Random
SINGLE.AREA:	True

The NDL programmer specifies the NDL file's external name, via the DCP TERMINAL statement. Each record of the file contains 180 bytes with each byte corresponding to an address in the DCP.

Known S-label Address File (SLBL.K)

This is an intermediate workfile containing all known S-label addresses. Its internal attributes are as follows:

INTERNAL FILE.NAME:	SLBL.K
MFID:	0000000
NO. BUFFERS:	2
RECORD:	180 bytes
BUFFER:	180 bytes
CLOSEMODE:	Release
FILETYPE:	@00@
ACCESSMODE:	Random
FILESIZE:	@000200@
FID:	SLBL,K

An S-label signifies a new S-level instruction. The SLBL.K file contains two elements for each S-level detected, as follows:

1. An S-level address.
2. The microaddress where the first microcode was generated for the S-instruction.

This file contains the S-labels for all request and line control procedures required for the program being generated. It is also used for:

1. Label resolution.
2. As a reference for microcode listing.

During resolution, this file supplies the microaddress of each referenced S-label. The resolution process searches the file for the S-label in question. When found, the microinstruction is inserted into the instruction that referenced the S-label.

When listing microcode, NPC refers to this file to determine where each S-instruction is to be printed. The print location is determined by sequentially accessing this file. NPC lists the microcode until detecting an address that has an S-instruction associated with it. NPC prints the S-instruction and continues listing microcode. This sequence continues throughout the listing process.

Unknown S-Label Address File (SLBL.UN)

This is an intermediate workfile containing references to unknown S-labels. Its internal attributes are as follows:

INTERNAL FILE.NAME:	SLBL.Un
MFID:	0000000
NO.BUFFERS:	2
RECORD:	180 bytes
BUFFER:	180 bytes
CLOSEMODE:	Release
FILETYPE:	@00@
ACCESSMODE:	Random
FILESIZE:	@000200@
FID:	SLBL.UN

This file contains two elements for each S-label reference to another S-label. The elements consist of:

1. The microaddress where the reference was made.
2. The S-label to which the reference was made.

The file is used during label resolution, and contains all labels that need to be resolved. It is maintained on a request set basis. After processing each request set, the labels are resolved and references to the file are re-initialized for the next request set.

Each logical record contains 45 references of four bytes each: two bytes for each S-level reference, and two bytes for the microaddress at the occurrence of the S-label reference.

NDL Object Code File (NDLSYS)

This file is input to the NPC compiler. It contains object code generated by the NDL compiler. Its internal attributes are:

```
INTERNAL FILE.NAME:      NDLSYS
MFID:                   0000000
FID:                   NDLSYS
RECORD:                 180 bytes
BUFFER:                 180 bytes
FILETYPE:               @10@
```

NPC accesses NDLSYS during the following three phases of execution:

1. The establishment process, to create tables required for generation.
2. Generation, to access S-instructions.
3. Listing, to include the S-instructions in the listing.

Printer Backup File (PBM)

The PBM file is an output file that stores information to be listed after code generation is complete. The internal attributes are:

```
INTERNAL FILE.NAME:      PBM
PMF ID:                  0000000
NO.BUFFERS:              2
CLOSEMODE:               Release
FILETYPE:                 @00@
RECORD:                  180 bytes
BUFFER:                   360 bytes
FILESIZE:                 @001000@
FID:                      PBM
```

Each file record is divided into nine 20-byte areas. Each generated microinstruction can place information into this file. The information may consist of the following:

1. An instruction label implying that the instruction is to be referenced by a GOTO or CALL instruction.
2. A location label implying that the instruction could pass control, for example, GOTO or CALL instruction.
3. A character string that gives a meaningful explanation to a literal associated with the instruction. The instruction usually has a literal value and is of the type TX = LIT. The LIT character string, when listed, could be documentary, for example, TX = LINE TABLE LENGTH.
4. A comment field that may contain any series of remarks to help document the intent of the instruction or series of instructions.

The file is created sequentially; that is, instructions are placed in the file in the order of generation. To access the file, a list of pointers indicates which information pertains to which instructions.

Printer Backup Key File (PBM.KEY)

PBM.KEY is an intermediate workfile that contains a list of pointers and indicators required to create the microcode listing. The internal attributes are:

```
INTERNAL FILE.NAME:      PBM.KEY
MFID:                   0000000
RECORD:                 180
BUFFER:                 360
CLOSEMODE:               Release
FILETYPE:                 @00@
FILESIZE:                 @000800@
FID:                      PBM.KEY
```

Each logical record contains 180 bytes divided into 60 three-byte elements. The elements are arranged by microaddress in ascending order. As the listing is created, the instructions are decoded (by reading M.CODE.FILE) and mnemonics are created in printable form. During this process, the PBM.KEY file is accessed to determine whether any additional information exists for the microinstruction.

The following explains the items found in each element:

1 ELEMENT (3 BYTES)	
2 INDICATORS (4 BITS)	This instruction has a label. The label name is in the PBM file.
3 MNEMONIC (1 BIT)	
3 MNEMONIC (1 BIT)	This instruction has a mnemonic associated with it. (Regardless of this flag's setting, there is nothing in the PBM.)
3 GO LABEL OR LIT LABEL (1 BIT)	This instruction either passes control (CALL or GOTO) or has literal values. In either case, the PBM file contains a 20-character string that is printed on the listing.
3 COMMAND (1 BIT)	This instruction has a comment for documentation purposes.
2 RECORD.NO (2BYTES)	Gives the byte index to the PBM file record where the indicated information starts.
2 RECORD.NO (2 BYTES)	Gives the PBM file logical record number where the indicated information starts.
3 DATA FIELD (1 BIT)	The instruction does not have a mnemonic; the generated item is data to be accessed by another instruction.

NOTE

A field indicator of zero indicates that this is a null entry and that there is no information in the PBM file for this microaddress.

PROGRAM FLOW

NPC flows from the initial establishment process, through microcode generation, and finally to the microcode listing phase. Each phase is described in the following paragraphs.

Establishment Process

Accessing the NDLSYS File

The establishment process sets-up the data for other phases of operation. It retrieves the information from the NDLSYS file and places it into post compiler memory. The disk files used during code generation are opened and made ready for use. The ESTABLISH procedure carries out the following tasks:

1. Retrieves and saves the addresses of the control and request sets and all tables required to generate object code.
2. Generates the following global tables in NCP memory for later use by the code optimization process: line, terminal, and station tables.

As NPC retrieves NDL tables, it builds global tables in NPC memory. The structure of global tables is similar to that of NDL tables. Global tables contain two types of entries:

1. Common entries: information that may be constant to the system.
2. Flagged entries: information that is not constant to the system.

As NPC retrieves each NDL table value, it compares the retrieved value against any existing values in the global table being built. If the value is consistent with existing values, NPC proceeds to retrieve the next value. If the value is different from an existing value, NPC flags that entry as "not-common" by placing the value @FF@ in the global table. For example, a global station table never contains the TRANSMIT or RECEIVE ADDRESS values because these are unique to the station. However, the global station table may contain the INITIATE RECEIVE value if the comparison process determines that it is a system-wide common value. If not common, the entry is flagged.

NOTES

1. Later, during code generation, NPC generates microcode for using the unflagged values. For flagged (uncommon) values, NPC generates microcode for both retrieving and using the value.
2. Also during code generation, flags in the global tables are handled on an ORed-in basis. If, for example, any one table has CASE SHIFT ON, the global table

has CASE SHIFT ON. If CASE SHIFT is not in the global table, NPC emits no code relating to case shift.

NPC retrieves and organizes data in post compiler memory by performing the following steps:

1. Open the NDLSYS file.
2. Read the Program Parameter Block (PPB) into memory. The PPB is the first record of the NDLSYS file. It contains, among other items, pointers to the Program Segment Table (PST) and the Data Segment Table (DST).

NOTE

All references to memory in this section indicate post compiler memory.

Following is a diagram of the PPB:

Field Name	Location	Length	Type	Contents
Implementation Level No.	0	1	Binary	@70@
Program Name	1-12	12	ASCII	"NDLSYS"
S-Language Name	13-24	12	ASCII	"NDL S-LANG"
Interpreter Pack-Id	25-31	7	ASCII	"0000000"
Interpreter Name	32-43	12	ASCII	"NDL INTERP"
Computer Name	44-55	12	ASCII	"NDL COMPILER"
Compilation Date	56-61	6	ASCII	YY MM DD
Priority Class	62-63	2	Binary	@3180@
Data Segment for Initiating Message	64	1	Binary	@FF@
S-Program Start Address	65-67	3	Binary	@000000@
Program Segment Table Length	68-69	2	Binary	@0030@
Program Segment Table Location	70-71	2	Binary	@0002@
Data Segment Table Length	72-73	2	Binary	@0072@
Data Segment Table Location	74-75	2	Binary	@0003@
TCB Preset Area Length	76-77	2	Binary	@0000@
TCB Preset Area Address	78-79	2	Binary	@0000@
Stack Length	80-81	2	Binary	@0000@
CCB Preset Area Length	82-83	2	Binary	@0000@
CCB Preset Area Address	84-85	2	Binary	@0000@
Tab Extension Length	86-87	2	Binary	@0000@
Internal File Name Block Length	88-89	2	Binary	@0000@
Internal File Name Block Address	90-91	2	Binary	@0000@
Tab Preset Area Values	92-179	88	Binary	All Zeroes

3. The PST segment is retrieved with the PST pointer located in the PPB. The PST segment contains pointers to each of the program parts (control sets and request sets). It is a series of descriptors (pointers) to actual code or displacement lists. The following is a diagram of the PST:

- Descriptor 0 - Control sets - Format A.
- Descriptor 1 - Control displacements - Format A.
- Descriptor 2 - Request sets - Format A.
- Descriptor 3 - Request displacements - Format A.
- Descriptor 4 - Control sets - Format B.
- Descriptor 5 - Control displacements - Format B.
- Descriptor 6 - Request sets - Format B.
- Descriptor 7 - Request displacements - Format B.

NPC uses the descriptors for Format A.

4. Using descriptor 1, NPC retrieves and saves (in NPC memory) the line control displacement list for later processing of line controls. This list of displacements gives the relative address within the code segment (pointed to by descriptor 0) where each logically-numbered control set begins. This segment also specifies the number of line controls within the program. This value is stored in the NPC variable PRESET.NLC.
5. Using descriptor 3, NPC retrieves and saves (in NPC memory) the request set displacement list for alter processing of request sets. The list of displacements gives the relative address within the code segment (pointed to by descriptor 2) where each logically-numbered request set begins. This segment also specifies the number of request sets within the program. This value is stored in the NPC variable PRESET.NREQ.
6. Using the pointer from the PPB, the post compiler retrieves the Data Segment Table (DST). This table has descriptors for each of the data areas of the program. The following is a diagram of the DST.

Descriptor 0 - Preset Data
Descriptor 1 - Line Tables
Descriptor 2 - Line Table Displacement List
Descriptor 3 - Station Tables
Descriptor 4 - Station Table Displacement List
Descriptor 5 - Modem Tables
Descriptor 6 - Terminal Tables
Descriptor 7 - File Tables
Descriptor 8 - Extended Station Tables
Descriptor 9 - Extended Terminal Tables
Descriptor 10 - Station Name Table
Descriptor 11 - File Name Table
Descriptor 12 - Translation Tables
Descriptor 13 - Translation Table Displacement List
Descriptor 14 - Line Priority Chart
Descriptor 15 - Line Speed Table
Descriptor 16 - DCP Terminals Format A
Descriptor 17 - Source Statement Occurrence
Descriptor 18 - DCP Terminals Format B

The pertinent descriptors are saved in NPC memory for alter access of the data areas.

7. NPC then retrieves the segment containing source statement occurrences. Descriptor 17 of the data segment table points to this segment. For each request set/line control there are two bytes of information specifying the occurrence of certain NDL source statements. A list of each statement represented in this area is given below. A bit within the appropriate two-byte area indicates each statement's occurrence within a request/control set. The statements that can be indicated are:

LINE.BUSY = FALSE
LINE.BUSY = TOGGLE
AUX LINE BUSY - FALSE
AUX LINE BUSY = TOGGLE
BINARY = TRUE/FALSE
TERMINATE BLOCK
SYNCS = TRUE/FALSE
CRC = TRUE/FALSE
SHIFT = UP/DOWN/MIDDLE
STATION = LIT/VARIABLE
IF LINE.CHAR//LINE.CHAR = LIT/VARIABLE// - VARIABLE = LINE.CHAR
RECEIVE (CONTINUE)
RECEIVE TEXT
BACKSPACE

NPC dynamically declares the SINST array (source statement occurrence data) by using the PRESET.NLC and PRESET.NREQ variables retrieved from the displacement lists for the line control and request sets. The array contains two bytes of information for each line control/request. NPC references this information throughout execution to perform optimization (based on the occurrence of the statements). NPC places data in the SINST array as follows:

- a. The line control occurrence lists are placed consecutively into the array by logical number starting at element zero.
- b. The occurrence lists for all the request sets are then placed consecutively by logical number following the line control occurrences.

As the placement occurs, an array is also created which contains the OR of all the occurrences for each line control/request. This array, G.SINST, is two bytes in length and represents the possibility of any particular occurrence throughout all line control and requests.

8. Using descriptor 0 of the data segment table, NPC retrieves the preset data segment. This segment contains, among other items, counts for each of the tables existing on the system, for example, the number of line tables. These counts are saved in post compiler memory as described below:
 - a. The number of terminals on the system is placed into the variable NO.LOGICAL.TERM. This variable is used frequently in the program for controlling loops on terminal types. This number is also used to declare the TERMINAL dynamic array to contain all terminal tables as declared in the NDL program. If the number of terminals is greater than a predetermined limit, the TERMINAL dynamic array is assigned to a data segment other than segment 0.
 - b. The number of lines on the system is used to declare both a dynamic array to contain all of the displacement lists for the line tables and the LINE dynamic array. The LINE array is large enough to contain all of the line tables for the system.
 - c. No table contains all of the stations on the system. Only the global station table is applicable.
9. Using descriptor 2 of the data segment table, NPC retrieves the line table displacement list.
10. Using the line table displacement list and descriptor 1 of the data segment table, NPC retrieves the line tables and places them consecutively by logical line number into the variable array, LINE.
11. Using descriptor 3 and the station table displacement list of the data segment table, NPC retrieves a portion of each station table and builds the global station table for the system. Because the individual station tables are of no use to the NPC at compile time, they are not kept in NPC memory.
12. Using descriptor 5 of the data segment table, NPC retrieves the terminal tables and places them into the variable TERMINAL. The tables are placed consecutively by logical terminal number.
13. Using descriptor 18 of the data segment table, NPC retrieves the DCP terminals segment. This segment tells the NPC which files are to be created and which terminals are on each file. Initially, NPC obtains a count of the number of files that exist. With this value, NPC declares an array which contains the file names for the files to be generated. The file count is also used to declare arrays containing the displacements for the lists of terminals for each file.
14. The file name, the number of terminals (PRESET.DCP.NTER), and the terminal displacement list are then retrieved for each file to be generated and placed into their respective arrays.
15. Using the terminal displacement list, NPC retrieves the list of terminals (logical terminal numbers) for each file and places them into the array DCP.TER.LIST.
16. NPC then sorts the list of terminals for each file by ascending logical terminal numbers.

The data required to generate all files identified by the NDL program has been properly placed into NPC memory. Later, some of the data, particularly the line and terminal tables, are re-arranged for the particular files.

GENERATING MICROCODE

This section details the following subjects:

1. Interrelated file establishment activities.
2. Translation of S-instructions into microcode.
3. Resolution of references to labels during code generation.
4. Generation of line disciplines and subroutines.
5. Types of storing.

In the process of generating the NDL named files, all S-level instructions are translated into microcode. The manager interrupt routines represent only the characteristics of the particular devices used. NPC performs the generation in the following manner:

1. Initialize FILE.CNT to zero. FILE.CNT is a variable used to index into the array of file names and to limit the number of files generated.

2. If FILE.CNT is equal to the total number of files named by the NDL program, discontinue program generation.
3. Locate the file name: DCP.FILE.NAME (FILE.CNT * 12).
4. If the file name is equal to NDLDLCP or BDLDCP, increment the FILE.CNT and return to step 2. (NDLDLCP and BDLDCP are reserved words and are not considered valid filenames for program generation.)
5. Open the code file, M.CODE.FILE, with the FID of this filename.
6. Save the filename for the microprogram listing.
7. Initialize variables (addresses, indexes, etc.) used during code generation.
 - a. Write a random number into the NDLSYS file preset area to be used for identification at load time.
 - b. Close NDLSYS.
 - c. Change MYUSE to input only.
 - d. Open NDLSYS.
8. Ascertain how many terminal types exist for this file and place the count into the variable TERMINAL.TYPES, that is, TERMINAL.TYPES : = PRESET.DCP.NTER (FILE.CNT);
9. Mark the LOGICAL.TERM.NO (first field of each element in the array TERMINAL) of all terminals with @FF@. This flag indicates that the particular terminal does not exist on this file.
10. For each terminal belonging to this file, place the file logical terminal number into the array LOGICAL.TERM.NO in each terminal table. This indicates that the terminal is valid for this file.
11. Using only the valid terminals for this file, reference the source statement occurrence data, SINST, and create the G.SINST table. G.SINST reflects the occurrence of source statements for all line control/request sets referenced in this file. Creating G.SINST is accomplished by simply ORing together all the pertinent fields.
12. Using only the valid terminals for this file, reference the terminal tables (TERMINAL) and create a global table (G.TER). G.TER reflects all the terminal characteristics for this file. Creating G.TER is accomplished by the following process:
 - a. Move the first TERMINAL data into G.TER.
 - b. If only one terminal exists, the process is complete.
 - c. Reference the next TERMINAL data.
 - 1) For bit-addressable fields, OR-in the field to the G.TER field.
 - 2) For other fields, (BYTE, FIXED, BIT(4)), compare the current TERMINAL data with G.TER data. If unequal, place @FF@ into the G.TER field. @FF@ flags the data as inconsistent among the terminals on this file; therefore, no optimization can be made.
 - d. Repeat step c until all terminals are completed. The resultant G.TER reflects:
 - 1) The occurrence of a particular option on the file for bit-addressable fields.
 - 2) The fact that certain values are constant or variant among all terminals for non-bit addressable fields.
13. Using all the valid lines for this DCP, reference the line tables (LINE), and create the global table (G.LINE). G.LINE reflects all line characteristics for this DCP. G.LINE creation is similar to that of G.TER. The post compiler determines if the line is multi-line or single-line by noting whether more than one line is defined for the DCP for which code is being generated. Higher throughput is obtained in a single-line environment.
14. Determine whether full-duplex is possible on this file by ANDing the full-duplex bits of G.LINE and G.TER. Also, generate G.FD.TER and G.FD.LINE if FD.POSSIBLE. (A file containing a full-duplex possibility is automatically a MULTI.LINE.FILE.)
15. Assign the data memory as follows:
 - a. Allocate space and place data for the reserved memory area.
 - b. Allocate space and place data for the terminal's translate tables. This is accomplished by:
 - 1) Using descriptor 12 of the data segment table, retrieve the translate table displacement list from the NDLSYS file.
 - 2) Using descriptor 11 of the data segment table with the appropriate displacement from the translate displacement list, retrieve the translate table for a particular terminal if it performs translation. The translate table is placed in the variable XLATE.TABLE in post compiler memory. The retrieval can cause up to four disk reads to occur.
 - 3) Once the table is located in post compiler memory, allocate microcode memory space for the receive side of the translation table. Place the receive side (every other character) into the code file. Allocate space for the transmit side of the translation table. Place the transmit side into the code file.

The absolute addresses of the translation tables are stored into arrays for later use by the interrupt logic. The array `TERMINAL.XLATE.RCV` contains all the addresses for the receive translation tables. The array `TERMINAL.XLATE.XMIT` contains all the addresses for the transmit translation tables.

Because the interrupt logic is tuned to handle specific terminals, it can reference the tables directly. Generating an instruction to address the base of the RECEIVE translate table appears as follows:

```
M1. (TERMINAL.XLATE.RCV (TERM.COUNT * 2));
```

M1 is a procedure call to generate a microinstruction.

`TERMINAL.XLATE.RCV` is the array name of the area containing the absolute addresses of the translate tables.

`TERM.COUNT` is an index for the particular terminal type for which code is currently being generated.

The resultant microcode is as follows:

```
M1 ← LIT
```

LIT is the absolute address of the base of the receive translate table.

The post compiler guarantees that duplicate translate tables are not loaded into memory. When loading a translate table, a check is made to ensure that this particular translate table has not already been loaded. If it has, the array items containing the address of the table are copied into the current array items.

16. Set up `IGNORE.FLAGS` array for this file. If it is determined that a terminal can be completely reduced, mark the `LOGICAL.TERM.NO` for that terminal in the `TERMINAL` array with `@FF@`.
 17. Generate all subroutines for the manager, interrupt handlers, and the host control function. Subroutines are generated only for terminals that are valid and not reduced.
 18. Generate the inline (non-subroutine) code for the manager, interrupt handlers, and host control function. The interrupt routines are terminal-dependent and generate one routine for each valid and not-reduced terminal type on this file.
 19. Initialize `TERM.TYPE.NO` to zero. `TERM.TYPE.NO` is a variable used to index into the terminals and to control the number of terminals on a file.
 20. If `TERM.TYPE.NO` is equal to the number of terminals on this file (`TERMINAL.TYPES`), skip to step 50.
 21. Retrieve the logical terminal number from the array of terminals for this file (`DC.TER.LIST`). Replace the variable `LOGICAL.TERM.TYPE.NO` with the logical terminal number.
 22. If the auxiliary line control is reduced, go to step 25.
 23. Get the displacement of this terminal's auxiliary line control (if it is a full duplex device). The displacement list is retrieved from disk during the establishment process.
 24. Make a pass through this terminal's auxiliary line control and generate all the subroutines it uses. The pass through the S-code is made by using the displacement to retrieve the instruction. This process fetches each S-level instruction, but generates only the microcode for the subroutines invoked.
 25. If the line control is reduced, go to step 28.
 26. Get the displacement on disk of this terminal's line control (if it is a full-duplex device).
 27. Make a pass through this terminal's line control and generate all the subroutines it may use.
 28. If the input request is reduced, go to step 31.
 29. Get the displacement on disk of this terminal's input request set (if one exists).
 30. Make a pass through the input request set and generate all the subroutines it requires.
 31. If the output request is reduced, go to step 34.
 32. Get the displacement on disk of this terminal's output request set (if one exists).
 33. Make a pass through the output request set and generate all the subroutines it requires.
 34. If the auxiliary line control is reduced, go to step 35; otherwise, go to step 36.
 35. Fill the auxiliary line control branch table entry for this terminal number. This number is indicated by the `IGNORE.FLAGS` array entry for this terminal's auxiliary line control. Go to step 38.
 36. Retrieve again the displacement on disk of this terminal's auxiliary line control.
 37. Make a pass through this terminal's auxiliary line control and generate all microcode that replaces the S-level instructions.
 38. If the line control is reduced, go to step 39; otherwise, go to step 40.
 39. Fill the line control branch table entry for this terminal with the address of the line control of the terminal number indicated by this terminal's line control entry in `IGNORE.FLAG` array. Go to step 42.
 40. Retrieve again the displacement on disk of this terminal's line control.
 41. Make a pass through this terminal's line control and generate the microcode that replaces the S-level instructions.
-

42. If the input request is reduced, go to step 43; otherwise, go to step 44.
43. Fill the input request branch table entry for this terminal with the address of the output request of the terminal number indicated by this terminal's output request entry in IGNORE.FLAGS array. Go to step 46.
44. Retrieve the displacement on disk of this terminal's input request set.
45. Pass through the input request set and generate the microcode that replaces the S-level instructions.
46. If the output request is reduced, go to step 47; otherwise, go to step 48.
47. Fill the output request branch table entry for this terminal with the address of the input request of the terminal number indicated by this terminal's input request entry in the IGNORE.FLAGS array. Go to step 50.
48. Retrieve the displacement on disk of this terminal's output request set.
49. Pass through the output request set and generate the microcode that replaces the S-level instructions. Resolve any unresolved S-labels.
50. Increment the variable to count the number of terminals (TERM.TYPE.NO) and go to step 19.
51. Resolve all references to the beginning of the line controls and request sets. These references are of the type generated in subroutines like INITIATE.REQUEST, where a request set may be entered.
52. Print the listing of the generated microcode.
53. Close the generated microcode file.
54. Increment the variable FILE.CNT to count the number of files.
55. Go to step 2 to generate code for any other files.

S-Instruction Processing

As the line control/request sets are processed, the individual S-instructions are translated into microcode. The types of translation can be broken into three general categories:

1. Those S-instructions that call subroutines that are invariant to the terminal type for that file.
Example: INITIATE REQUEST. This S-instruction generates a CALL instruction to the common subroutine that performs all initiate requests. The subroutine handles all possible terminal types existing within the particular file. Subroutines may be tuned in some instances but the optimization is of a general nature; for example, the routine may check data variables such as TERMINATE BLOCK, which indicates whether or not that instruction was ever used. If never used, NPC does not generate microcode to perform that subroutine.
2. Those S-instructions that call subroutines that are tuned for the terminal type.
Example: INITIATE TRANSMIT. This S-instruction calls a subroutine that is tuned for the terminal referencing the request set. The post compiler also determines which label within the subroutine is pertinent. Consequently, an INITIATE TRANSMIT subroutine exists for each terminal type within the file.
3. Those S-instructions that do not access subroutines and are self-contained within the request/control microcode.
Example: TOG[1] = TRUE. Most S-instructions that access variables use a routine to load or store the data variable from or to its location.

A format is associated with each S-instruction. Some instructions have a fixed length, while others are variable in size. The length of the variable instructions is determined by particular fields within the instruction. The right byte of the first word indicates the instruction Op-code. The error branch switches occur at the end of each request set. They are in groups of six branches. Each group can be referenced by the receive instructions, with the branches corresponding to the six error flags obtainable during a RECEIVE.

S-Instruction Processing Examples

The four S-instruction processing examples presented in this section are as follows:

- Example 1: ADD.VALUE.VALUE
- Example 2: ASSIGN.TOGGLE.TO.TOGGLE
- Example 3: PAUSE
- Example 4: INITIALIZE.TRANSMISSION.NO

Each example is a general case of translation from S-level to microcode. These and subsequent examples illustrate post compiler decisions and the resultant generated microcode. The microcode presented is the actual mnemonics for the code placed in the codefile for each example.

NOTE

In these examples, the generated microcode is contained within brackets so the reader can distinguish the microcode from the MPLII code.

Example 1

ADD.VALUE.VALUE

X(1)	OP CODE
X(2)	X(3)

$$X(1) + X(2) \longrightarrow X(3)$$

This S.OP adds the contents of the byte variables X(1) plus X(2) and places the result in byte variable X(3). The following is the general flow of the S.OP code emission:

PROCEDURE ADD.VALUE.VALUE;

1 IF X1 = X2 = X3 THEN % All operands specify the same byte variable (B.V.).

DO EMIT;

```
[M1 ← B.V. ADDRESS]      % Set up memory address of B.V.
[B0 ← N1]                 % Read memory into B0 without
                           % changing memory address register
[WRL ← B0]                % copy value into WRL.
[B0 ← B0 + WRL]           % X(3) ← X(1) + X(2)
[I1 ← B0]                 % Memory address register still
                           % points to B.V. so write B0 to
                           % memory.
```

END EMIT;

1 IF X1 = X2 THEN DO; % X3 2 * X1

2 1F X1 AND X3 IN SAME TABLE THEN DO;

3 1F X1 AND X3 CONSECUTIVE BYTES THEN

DO EMIT;

```
[M1 ← B.V. ADDRESS (OF X1)]
[B0 ← I1] % Read B.V. from memory and increment
           % memory address register. M1 now
           % points at X3.
[WRL ← B0] % Copy B.V. to WRL
[B0 ← B0 + WRL] % B0 ← 2 * B.V. (X1 + X2)
[I1 ← B0] % Store result in X3
```

END EMIT;

3 1F X3 AND X1 CONSECUTIVE BYTES THEN

DO EMIT;

```
[M1 ← B.V. ADDRESS (X1)]
[B0 ← N1]
[M1 ← M1 - 1] % Decrement memory address register
               % to point at B.V. address (X3)
[WRL ← B0]
[B0 ← B0 + WRL] % X1 + X2
[I1 ← B0]
```

END EMIT;

END 2;

```

DO EMIT;                                     % X1, X3 No relation
  [M1 ← B.V. ADDRESS (X1)]
  [B0 ← I1]
  [ WRL ← B0]
  [B0 ← B0 + WRL]                             % X1 + X2
  [M1 ← B.V. ADDRESS (X3)]
  [I1 ← B0]
END EMIT;
END 1;
1 IF X1 = X3 THEN
  DO;                                         % Similar to above
  .
  .
  .
END 1;
1 IF X2 = X3 THEN
  DO;
  .
  .
  .
  END 1;
DO ALL.BYTE.VAR.DIFFERENT;
.
.
.
END ALL.BYTE.VAR.DIFFERENT;
.
.
.
END ADD.VALUE.VALUE;

```

This procedure attempts to find where each of the parameters for the S-instruction are located and to arrange the microcode based on these locations. Generally, all byte-variables are processed in this manner.

Example 2

ASSIGN.TOGGLE.TO.TOGGLE

TOG-1-A	OP CODE
TOG-1-C	TOG-1-B
	TOG-2-A
TOG-2-C	TOG-2-B

This S.OP replaces the value of toggle 2 (denoted by TOG-2-A, TOG-2-B, and TOG-2-C) with the value of toggle 1 (denoted by TOG-1-A, TOG-1-B, and TOG-1-C).

Toggle addressing comprises labels A, B, and C, where:

A = Bit mask.

B = Table pointer.

C = Pointer to the byte within the table.

Therefore, if TOG-1 = TALLY [0] [4], then:

TOG-1-A = @(1)00010000@

TOG-1-B = STATION TABLE.

TOG-1-C = TALLY[0].

The following is the general flow of the code generated for this instruction:

```

PROCEDURE ASSIGN.TOGGLE.TO.TOGGLE;
[M1 <--- ADDRESS OF SENDER]                                % Decoded by using
                                                            % TOG-1-C, TOG-1-B
[B1 <--- N1] % BYTE CONTAINING TOG TO B1
IF TOG-1-B = TOG-2-B AND                                    % Notice whether or not
TOG-1-C = TOG-2-C                                         % toggle is being replaced
THEN                                                       % by another in the same
[B0 <--- N1]                                               % byte
ELSE
  [M1 <--- ADDRESS OF RECEIVER;]
  [B0 <--- N1;]
  [B1 <--- B1 OR (TOG-1-A FALSED)]                          % Is sender TRUE or FALSE
  [IF KS GO TO SET.TOGGLE;]
[RESET.TOGGLE]
[B0 <--- B0 AND (TOG-2-A FALSED);]
[GO TO WRITE.TOGGLE;]
[SET.TOGGLE:]
[B0 <--- B0 OR (TOG-2-A)]
[WRITE.TOGGLE:]
[I1 <--- B0;]
END ASSIGN.TOGGLE.TO.TOGGLE;

```

Example 3

PAUSE

	OP CODE
--	---------

The PAUSE instruction causes control to be relinquished by the executing line to allow servicing on other lines. Code generation for PAUSE is as follows:

```

PROCEDURE PAUSE;
IF MULTI-LINES:
  THEN                                     % Relinquish control
    [CALL PRE-MANAGER.PAUSE]
  ELSE                                     % Single-line-case
    IF A.LINE.BUSY = FALSE COULD HAVE BEEN EXECUTED
    OR
    IF A.LINE.BUSY = TOGGLE COULD HAVE BEEN EXECUTED
    THEN
      [CALL HOST-CONTROL]                 % Check for any host interrupts
END PAUSE;

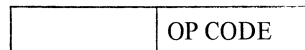
```

This procedure determines whether or not the configuration is multiple lines. If it is, an instruction is generated to enter the manager and perform a PAUSE on this line. If the configuration is single-line, PAUSE has no meaning unless LINE-BUSY had been reset. If LINE-BUSY has not been accessed, no code is generated. If LINE-BUSY had been reset, a call is generated to look for host interrupts.

This example shows the microcode that could be generated. Notice that the decoding of the toggles is done by the post compiler so that an instruction (MI ← ADDRESS OF SENDER) could be generated to directly reference the particular toggle. Notice also that the post compiler checks whether the toggles are both in the same byte, and if so, does not generate instruction(s) to reference the receiver's address.

Example 4

INITIALIZE.TRANSMISSION.NO.



The microprogram instruction places zeros into the transmission number as follows:

1. In the receive transmission, if in an input request.
2. In the transmit transmission number, if in an output request.

The following illustrates the general flow in generating microcode for this instruction:

```

PROCEDURE INITIALIZE.TRANSMISSION.NO;
                                     % Emit code to address the STN.TABLE
[M1 < -- K]
IF INPUT.REQUEST.SET
  THEN                               % Address the RC-TR#
    [M1 < -- M1 + STN.RCV.TR.NO.]
  ELSE                               % Address the XMT-TR#
    [M1 < -- M1 + STN.XMT. TR.NO.]
% EMIT.CODE TO ZERO THE VARIABLE
[M1 < -- 0]
[M1 < -- 0]
END INITIALIZE.TRANSMISSION.NO;

```

This example displays the possible testing of some states at post compiler time. The procedure that generates the code for INITIALIZE.TRANSMISSION.NO notices that the microcode resides in either the input or output request set.

Label Resolution

Three types of labels exist during the creation of a microcode file:

1. S-labels: labels resulting from S-instructions.
2. M-labels: labels associated with routines within the microcode.
3. Labels associated with entry to a request/control set.

Each label type is described in succeeding paragraphs.

S-label Generation

S-instruction labels (S-labels) are always resolved at the completion of a line control/request set. The address (microlevel and S-level) of each S-instruction encountered (considered to be known-labels that can be referenced) are saved. They are placed in sequential order (by S-label) in a disk file.

Each reference (unknown-label) to an S-label by an S-instruction also has its microaddress (and the S-address it is referencing) saved. At the end of each request or control, unknown S-labels are resolved by indexing into the known labels with the unknown S-labels and retrieving its microaddress for insertion into the unknown S-label's microaddress.

The following is an example of processing S-labels.

```
NDL Statements:
    GETSPACE [3]
    .
    .
    .
3: TERMINATE ERROR.
```

As each new S-instruction is encountered, its micro and S-label addresses are saved. The microaddress (occurrence) of the GETSPACE instruction and the TERMINATE ERROR are saved with the instruction, and the TERMINATE ERROR is saved with the S-level address of each. The S-level address and the microlevel occurrence are maintained in a disk file. The occurrence is any known label that may or may not be referenced by some other NDL instruction. The microcode for the NDL instruction is generated, including the GOTO that invokes the logic specified by the NDL programmer, via a NO SPACE condition (LABEL [3]). The label for the NO SPACE branch is not known and is saved for later resolution. The saved information is the address of the GOTO (microaddress) and the label address (S-level).

As other instructions are processed, the known labels are saved. When the control/request set is completely processed, the labels that were not previously resolved are then resolved.

M-label Resolution

NPC resolves microinstruction labels in two ways:

1. By generating the called label when it is first referenced.
2. By saving the microcode reference for later resolution.

To resolve a label at its first reference, NPC uses the function name of the subroutine containing the label. This name then becomes the attribute in a CALL/GOTO statement. The label is generated and the address is returned before the CALL or GOTO is generated. Any later references to the same label cause the

function to return the address of the label. For example, a statement to generate a CALL to a subroutine GETSPACE appears as:

```
MCALL(GETSPACE(GETSPACE.TERMINATE));
```

The following defines each element of the previous statement.

MCALL

The procedure that generates the actual microcode CALL instruction.

GETSPACE

The function that generates the microcode subroutine to perform a run-time GETSPACE.

GETSPACE.TERMINATE

The define that names the label within the GETSPACE routine to which execution is to be transferred.

The value of GETSPACE.TERMINATE is passed to the function GETSPACE which generates the microcode subroutine and returns the address of the label requested (GETSPACE.TERMINATE). The address is then passed to the procedure MCALL to generate the instruction with the correct address of GETSPACE.TERMINATE.

When the second form of label resolution is being used, the reference is saved for later resolution. This mechanism is used for references to labels within the same subroutine or labels in the interrupt routines and host control.

NPC supplies an array label identifier of the referenced label and a function call to the CALL or GOTO procedure. This call saves the reference. The M.LABEL function, for example, determines whether or not the label is already known. If not known, the reference is saved for later resolution. If known, the function merely returns the microaddress of the label.

The statement to call the manager to perform a receive appears as:

```
MCALL (M.LABEL(LABEL.PRE.MANAGER.RECV + TERMINAL.NO));
```

The following defines each element of the previous statement.

MCALL

The procedure that generates the actual microcode CALL instruction.

M.LABEL

The function that detects whether or not the label has a corresponding address. If no corresponding address, a value of zero is returned to generate a CALL; the reference is saved until it is resolved. If the corresponding address is known, the value is returned to emit the CALL instruction with the proper address.

```
(LABEL.PRE.MANAGER + TERMINAL.NO)
```

This is the label referenced for the particular terminal type which is passed to M.LABEL.

Control/Request Label Resolution

Labels associated with entry points to request or control sections are resolved upon completing the generation of the entire file. To place an instruction which would enter a request or control section, the following type of statement is used:

```
MGOTO (CONTROL.REQUEST.ADDR(PROTOCOL.TYPE, TERMINAL.TYPE))
```

The following defines each element of the previous statement.

MGOTO

A procedure that generates the microinstruction GOTO.

CONTROL.REQUEST.ADDR

The function that saves the reference and returns a value of zero.

PROTOCOL.TYPE

This is the section of that terminal's discipline being referenced. Legal attributes are:

INPUT.REQUEST
OUTPUT.REQUEST
LINE.CONTROL
AUX.LINE.CONTROL

TERMINAL.TYPE

This is the terminal's number of the request or control set being referenced. Note that this is the DCP relative terminal number (DRTN). In an S-OP, DRTN is stored in the variable *TERMINAL.TYPE.NUMBER*.

At the beginning of the generation of each request or control, the micro-address is saved with its terminal type number. At the end of processing this file, all references made through this mechanism are resolved. This is done by placing the address of the entry to the particular part of the discipline into the GOTO that was generated.

The previous structure is used in an S-OP to resolve request or control addresses.

For subroutines a different method is employed. Since the subroutines are general, all possible terminals are handled. Furthermore, the DRTN (DCP relative terminal number) is not stored in any run-time tables, but the logical terminal number is kept in the line table. The problem, therefore, is to relate the logical terminal number to the DRTN before invoking *CONTROL.REQ.ADDR*. The following scheme is used:

1. An index table is planted in the code file that relates all logical terminals in NDLSYS to a DRTN. (If a particular terminal is invalid on this DCP, it is given a DRTN of the last legal DRTN + 1.)
2. Branch tables containing the addresses for all valid DRTNs are planted on a per-procedure basis, for example, LINE CONTROL, AUX LINE CONTROL, XMIT REQ, RCV REQ. In case an invalid terminal should somehow execute, the last legal DRTN + 1 on entry is always the address of TERMINATE ABORT.
3. If a particular procedure does not exist for a terminal, the branch table contains the address of an *NDL.ERROR* routine. This procedure stops execution on the DCP and sets the ERROR byte to two.

Line Discipline

S-instructions are translated directly into microcode. Each S-instruction encountered has a direct correspondence with a portion of microcode. S-level instructions may also cause numerous subroutines to be generated. Generally, the largest portion (and most variant in size) of code generated results from translation of the S-instructions.

Subroutines

A subroutine is a portion of code that is used repeatedly during the execution of the code file. Subroutines are generated as a result of their first reference so the codefile contains only those subroutines needed by some portion of the code. A subroutine must be general enough to handle any terminal in the codefile, and is tuned on a file-basis rather than a terminal-basis.

The NPC maintains all microsubroutines as MPL functions that return the microaddress of that particular subroutine. These addresses are kept in a LABEL.ADDRESS array. When a subroutine is invoked, a test is made on the pertinent entry in the LABEL.ADDRESS array. If the address is zero, the subroutine must be generated. If the address is non-zero, the subroutine has previously been generated. In either case, the microaddress of the subroutine is returned to the invoking statement.

Subroutine Generation

Subroutine generation is accomplished by a two-pass system which proceeds as follows.

NPC makes two passes of all non-subroutine code (S-OPs, manager, and host control). The first pass causes all subroutines to be generated. The second pass emits all non-subroutine code. Because the subroutines are generated in the first pass, the subroutine addresses are already resolved by the time their first call is generated.

Because a subroutine may require other subroutines, each subroutine makes two individual MPL passes. The subroutines' first pass generates no code, but merely travels the same paths to be generated by the second pass.

The first pass finds all required nested subroutines. Each nested subroutine, in turn, enters its two-pass generation cycle. At some point, a subroutine having no further subroutines completes both passes. This control is returned to the invoking subroutine for completion of its passes. Subroutines are emitted in reverse of their nesting order and each subroutine's address is resolved before its code is generated.

Subroutines that reference one another cannot be generated in the previous manner because they cause an infinite loop. NPC forces referenced subroutines to the code file before generating any other code. This ensures that referenced subroutines are resolved before any other code is generated.

Example

A typical example of subroutine generation is:

```
MCALL.JPLUS (TERMINATE.NORMAL(TERM.NORM.LINE.BUSY));
```

This MPL source statement could be encountered in the procedure that processes the S-instruction TERMINATE NORMAL. Within the first pass of the S-OP procedure, it is determined that a call be made to the TERMINATE.NORMAL subroutine and at the label TERM.NORM.LINE.BUSY. The function that generates the terminate normal subroutine appears as follows:

```
FUNCTION TERMINAL.NORMAL(ENTRY.LABEL);
IF LABEL.ADDR(LABEL.TERM.NORM.LINE.BUSY) = 0 THEN:
MAKE 2 SUBROUTINE PASSES
  1. NON GENERATING
  2. GENERATING
  .
  .
  .
END SUBROUTINE GENERATION
CASE ENTRY.LABEL;
RETURN LABEL.ADDR (LABEL.TERM.NORM.LINE.BUSY)
RETURN LABEL.ADDR (LABEL.TERM.NORM.GETSPACE)
  .
  .
  .
END CASE;
```

The data item LABEL.ADDR is an array that contains all the addresses of the subroutine labels. The identifier LABEL.TERM.NORM.LINE.BUSY is an identifier for a define that names the array location of this subroutine's address. In effect, comparing for zero indicates whether or not the subroutine has been generated. If it is zero, the function proceeds by beginning its non code-generating (first) pass. Once the first pass is complete, code generation for this subroutine begins.

The current micro is assigned to the LABEL.ADDR array in the place reserved for the subroutine label. Thus, the subroutine is flagged as emitted. Individual microinstructions are then emitted. As each microinstruction is generated, the current address pointer (MICRO.INDEX) is incremented to the next location. If the function is later invoked, the compare on the array element indicates that the subroutine has been generated.

The address of any label occurring within the subroutine is saved into the array LABEL.ADDR and might appear as:

```
LABEL.ADDR(LABEL.TERM.NORM.GETSPACE) := MICRO.INDEX;
```

Upon completion of the function, the proper address is returned by determining which label was referenced by ENTRY.LABEL (the formal parameter).

Example

The value is returned to the statement that caused the function but was not used during the first S-OP pass. The second S-OP pass finds the subroutine already generated. Once again, the subroutine's function returns the address. This time that address is used as the address required by the microinstruction call J + <address>.

Microcode Space Overflow Detection

Because the amount of space required by the firmware programs varies, each program generated must not exceed allowable DCP space.

To monitor the amount of memory firmware programs require, each instruction emitted causes NPC to check for an overflow condition. Overflow is detected by comparing the current store address (MICRO.INDEX) with the maximum amount of memory (MEMORY.SIZE = 65K) available on a DCP.

When the codefile is completely generated, NPC calculates the size of the area needed to store data (line information, station tables, and so on). NPC then adds this area to the codefile size and checks the DCP to see if there is sufficient space to load the file. If not, an error is monitored.

Types of Storing

There are two types of storing: microcode and data.

Microcode Storing

Instructions are stored in post compiler memory in a disk buffer. When this disk buffer is full or another disk area is required, the instructions are written to the M.CODE.FILE disk file where they are stored one byte at a time. The following is a list of actions that occur as each byte is being stored:

1. Ensure that there is enough space (DCP memory) to perform a store.
 2. Place byte in the appropriate disk buffer, causing the following to occur:
 - a. Check whether the disk buffer in memory is the buffer needed to store this instruction.
 - b. If it is the desired buffer, store the instruction and proceed to step 3.
 - c. If not the desired buffer, perform the following:
 - 1) Write buffer in memory to disk.
 - 2) Set an indicator to specify that there are instructions in that buffer.
 - 3) Check indicator for the required buffer.
 - 4) If indicator is set, bring buffer into memory from disk file.
 - 5) If indicator is not set, initialize buffer area in memory to zeros.
 3. Decode the instruction so that pertinent information is placed in the printer backup file.
-

4. Increment the instruction counter that specifies the current address where code is being generated.

To store an individual microinstruction, the following sequence takes place:

1. A procedure is invoked through a series of defines. This procedure supplies the particular Op code for an instruction. There is one define for each instruction. For example, AD.(LIT) is defined as #EMIT.LIT1 (@38@,LIT)#. This define generates an instruction of the form: AD ← LIT;. This passes the appropriate parameters (@38@,LIT) to the invoked procedure (EMIT.LIT1).
2. The procedure places a call to the MICRO-SPACE function with the number of bytes to be stored. Two bytes are stored for AD ←LIT. MICRO-SPACE ensures the availability of at least the number of bytes specified as follows:
 - a. If there is sufficient space, a TRUE value is returned and the parameters for the instruction are placed into global variables OP1, OP2, and OP3. If the instruction contained three bytes, this procedure byte-reverses the second and third bytes as required by the processor.
 - b. If there is insufficient space, a FALSE value is returned indicating that the store should not be performed. The ABORT-FILE flag is set so that after this series of microcode is generated, code file generation is terminated.
3. The MICRO-STORE procedure then performs the actual store. MICRO-STORE calculates the displacement into the disk buffers. In the calculation process, the program may determine that the proper disk record is not in memory. The program then performs a write of the record in memory and a read of the desired record. Once the displacement has been calculated, a byte of the instruction is placed into the disk buffer area. If more bytes exist for the instruction, the displacement is again calculated for each byte.
4. After all bytes have been properly stored, a call is placed to the PBM.DECODE procedure. This procedure saves information about the instruction for listing purposes. The information could consist of a comment, a label to which the instruction might transfer execution, an associated label that other instructions might reference, and/or a literal value an instruction could use.
5. The program increments (by the number of bytes stored) the variable representing the current generation location in memory.

Data Storing

Data is stored in primarily the same fashion as instructions. The major difference between data and instruction storing is the appearance of the defines to which parameters are supplied. There are two defines for storing data items:

1. PLANT.D stores one byte of data at the current address, MICRO-INDEX.
2. PLANT.ADDR stores two bytes of data byte-reversed. This is used for storing addresses only.

MANAGER SCHEMES

Two types of managers can be generated from the post compiler. One is used for multiple lines (multi-lines), and the other is used for a single-line configuration. The interrupt routines for each go through identical types of optimization for the particular terminal types. The methods of invoking the interrupt routines at micro-code run-time differ as follows:

1. The multi-line version has a common routine that invokes the particular function the current line is performing.
2. The single-line version invokes the interrupt routines directly from the request/control logic.

Succeeding paragraphs describe both cases with an in-depth discussion of multi-line and a comparative analysis of the single-line case. Pre-managers are generally used to set-up entry points to the various manager types before yielding processor control to other lines on the system.

Multi-Line Manager

The multi-line manager consists of five basic functions: manager entry, manager line change, priority entry, priority line change, and manager exit.

The multi-line manager uses a “top down” queue of all active lines on the DCP. When a line is made ready, it is inserted in the top down line queue using the priority code found in the station table for relative station 0. Higher speed lines have a higher priority code and are at the top of the queue. Each line in the queue has a pointer to the highest priority line. The only exception to this is if there are several lines having the same priority and that priority is the highest on the DCP. In this case, these lines behave in a “round robin” fashion.

Manager entry is entered from the various pre-manager routines, except the XMIT/RCV CHAR/TEXT pre-managers, which go to priority entry. Manager entry stores the return address of the calling routine that invoked the appropriate pre-manager. The return pointer is popped off the hardware stack and placed into LNE.RETURN.POINTER in the line information area. The manager is always entered with only one address in the stack; that is, normally the address from which it was called in the line discipline. If the discipline has invoked a subroutine that calls the manager, that subroutine saves the address of the return point in the discipline in the variable LNE.SAVE.PTR. When control returns to the subroutine, the subroutine restores the line discipline’s return address and returns control to that address.

Manager line change is entered either from a manager entry or from interrupt handlers (managers) that are waiting for an interrupt. The function of manager line change is to pass control of the processor from the current active line to either the next line in the “top down” queue or to host control. Host control is treated as another line in the system. When passing control to the next line, the L-register is set-up to contain the base address of the next line’s line information area (LNE.NEXT.POINTER). This is used to set-up the next line’s PORT.NUM in the address register. Manager line change then does a GOTO to the next line’s current function type.

The priority entry and priority line change routines are used by character handling routines. Whenever a XMIT or RCV character has been processed by a line, that line goes to the priority manager code which switches to the highest priority line. This switch causes top down handling of XMIT/RCV characters. When a line is waiting for an interrupt, it goes to the next lower priority line using the normal manager line change routine.

Manager exit is entered when the current function type requires a return to the procedure that invoked the manager. Manager exit sets up the K-register to point to the current active station pointer, and then performs the return.

Succeeding paragraphs describe the various types of manager/pre-manager routines used in conjunction with the multi-line manager.

Idle

The pre-manager idle routine sets-up the response for an idle of a particular line. This is a direct result of the NDL IDLE S-OP.

The idle routine enters into the line’s function variable, LNE.FUNCTION, the address where control is passed when this line is next-selected. The manager type for idling a line is that of manager line change.

When the line is selected, control passes to the specified address and effectively bypasses that line. Manager line change selects a new line. Therefore, the line is ignored until a request that changes its state is honored.

Pause

Pause allows other lines to be serviced in a multi-line environment. This can be the direct result of a PAUSE statement in NDL or via implicit pauses within the firmware. Implicit pauses may be caused by:

1. GETSPACE.
2. Termination of a request set through one of the terminate instructions.
3. INITIATE REQUEST.
4. INITIATE ENABLE INPUT.

The function of the pre-manager pause is to write the manager exit address to memory (the LNE.FUNCTION variable). Control is then passed to the next line in the line queue. This allows one pass of all the lines to be made and then returns immediately to the paused line. This line exits the point that invoked the PAUSE.

Delay

Delay is a means to pause the line for a specified time. The delay can be a direct result of the NDL DELAY statement or through implicit delays within the firmware such as:

1. INITIATE RECEIVE.
2. INITIATE TRANSMIT.
3. BREAK INSTRUCTION.
4. FINISH TRANSMIT.

The pre-manager delay writes MANAGER.DELAY.ADDRESS to memory. Control is passed to the next line. When the delayed line is reselected, control passes to the MANAGER.DELAY.ROUTINE. It checks the adapter timer flag. If the timeout is reached, the routine returns to the request set at the point where the delay was invoked. There is a pre-manager and a manager for each of the two timers in the adaptor.

Receive

NPC generates a pre-manager receive routine for each terminal type in the system. This routine sets-up the timer and writes the address of the receive interrupt handler to line function variable, LNE.FUNCTION. There is also a receive interrupt routine tuned for each particular terminal type.

During the generation of the interrupt handler, characteristics of the terminal are considered.

Transmit

NPC generates a separate pre-manager transmit routine for each terminal type in the system. A single transmit interrupt routine is emitted for all terminals. PRE.MANAGER.XMT is responsible for:

1. Writing the address of the interrupt handler (LNE.FUNCTION) to memory.
2. Preparing the character for transmission, that is, vertical parity, translation, and so on, if necessary.

Full-Duplex Managers

When one half of a full-duplex line causes an action on its co-line (for example, FORK), a full-duplex manager completes that action. The causing half schedules the appropriate full-duplex manager for its co-line to accomplish this task.

Single-Line Manager

The single-line manager routine differs from the multi-line version in several ways:

1. No line change routine exists.
2. The interrupt routines do not return to a common place if their task cannot be carried out. Since this is the only active task within the system, the specific interrupt handler loops until the condition is satisfied, or until an error condition is detected.
3. Interrupt routines call host control while they are waiting for an interrupt.

CODE OPTIMIZATION

To enhance throughput, NPC tunes the microcode in the DCP to the existing configuration. Succeeding paragraphs are examples of how NPC performs this optimization.

Terminal Types

NPC optimizes the configuration using the terminal names that a particular file handles as specified in NDLSYS.

The interrupt routines, which play a critical role in throughput, are generated to handle only those characteristics that exist within the named file. In particular, NPC optimizes the transmit and receive interrupt handlers by creating a separate handler for each terminal type. Factors used for optimization are:

TRANSLATION
SYNCHRONOUS or ASYNCHRONOUS
CASE SHIFT
VERTICAL PARITY
SUMMED PARITY
PARITY MASK
HORIZONTAL PARITY
BCC or CRC HORIZONTAL PARITY ACCUMULATION
CRC-0 or CRC-1
CHARACTER SIZE
TRANSPARENCY
VERTICAL ODD/EVEN PARITY
SYNC CHARACTER
SYNC TRUE/FALSE
CRC TRUE/FALSE

Terminal Code Reduction

When two or more terminals possess similar characteristics, a common copy of the terminal dependent routines are generated.

NPC first compares the following characteristics for each terminal type:

PARITY (Vertical or Horizontal)
TRANSLATION
BCC INITIALIZATION (Zeros or Ones)
FULL DUPLEX
TRANSPARENT
CASE SHIFT
BCC/CRC
SUMMED PARITY
CRC-1 (CRC Polynominal)
SYNCHRONOUS/ASYNCHRONOUS
NUMBER OF TRANSMISSION CHARACTERS
NUMBER OF TRANSMIT ADDRESS CHARACTERS
NUMBER OF RECEIVE ADDRESS CHARACTERS
SYNC CHARACTER
PARITY MASK (NUMBER OF BITS NOT INCLUDING PARITY)
BITS
BDI
TELEX
ASCII/EBCDIC SYNCH CHARACTERS
SPECIAL TYPE
MAX INPUT VALUE

If two or more terminals satisfy the previous requirements, the addresses of all line control and request sets are compared. For each terminal type having a line control or request set equal to a generated terminal type, NPC places the terminal number in the IGNORE.FLAGS array entry for the generated terminal at the respective line control or request set offset.

If all line control or request sets for this terminal have been generated, the LOGICAL.TERM.NO field of the terminal array is set to @FF@. This indicates to the manager and to the terminal-dependent code not to emit code for this terminal. When generation of line control and request sets takes place, the address of the generated line control and request sets indicated by the IGNORE.FLAGS array are placed in the branch tables entry for the terminal.

The new result is a reduction in the amount of terminal dependent microcode in a particular file for line control, request sets, terminal-dependent manager and pre-manager, and the adapter. This increases the likelihood that a large codefile will fit into DCP memory. It does not, however, affect throughput.

DCP-Related Optimization

During codefile generation, the NCP utilizes certain DCP advantages such as those related to terminal information. Also, NPC uses the DCP adapter and DCP registers to assist in generating code. Use of these components is described in succeeding paragraphs.

Adapter

The adapter generates the vertical parity for each character during transmission except under the following conditions:

1. When the terminal has both vertical and summed horizontal parity.
2. When TRANSPARENT = TRUE/FALSE in the control/request set.

If these conditions exist, firmware generates the parity.

Registers

Throughout processing, NPC takes advantage of the variables in each DCP register, notably during processing of byte-variables. If the next sequential instruction to be processed requires the stored variables, NPC generates the necessary code.

In processing the S-instruction for:

```
TALLY[1] = TALLY[1] + 1
```

the generated microcode is as follows:

```
M1 ← K           % station table base
M1 ← M1 + tally offset
B0 ← N1          % tally 1 to B0
B0 ← B0 + 1      % increment
I1 ← B0          % store
```

The same byte variable is referenced in the replacement; consequently, the tally is loaded such that M1 points to its location after the load. When added, the literal value is stored directly without computing the address.

NDL Compiler-Related Optimization

As previously stated, NPC uses NDLSYS terminal information to optimize code. Additionally, NPC uses the NDL compiler-supplied information about the following S-level instructions to optimize certain areas:

```
ASSIGN LINE BUSY
LINE BUSY FALSE
ASSIGN AUX LINE BUSY
AUX LINE BUSY FALSE
TRANSPARENT = TRUE/FALSE
TERMINATE BLOCK
SYNC = TRUE /FALSE
CRC = TRUE/FALSE
SHIFT = UP/DOWN/MIDDLE
STATION = LIT/VARIABLE
LINE CHAR REFERENCED
RECEIVE WAIT S.OP OCCURS
RECEIVE TEXT S.OP OCCURS
BACKSPACE IS REFERENCED
```

This information is used during generation to optimize in certain areas.

Line Busy Information

During generation, NPC checks line-busy information for indications that the LINE BUSY flag has either been set FALSE or was falsed by replacement with another flag.

If the flag has never been set FALSE, NPC eliminates code in routines (such as INITIATE REQUEST, INITIATE ENABLE INPUT, and all TERMINATE statements) that implicitly or explicitly set this flag TRUE.

If the system is a single-line system, the PAUSE statement also checks the LINE BUSY information.

If the flag has never been set FALSE, the PAUSE is ignored for the single-line case. Effectively, the PAUSE has no meaning in a single-line system unless the request set is intended to be terminated through a host control request.

TRANSPARENT = TRUE/FALSE

The TRANSPARENT flag is used during generation of the manager. Normally, the manager needs to check the TRANSPARENT flag on each receive and transmit to determine its path. NPC uses this information to determine whether or not to test the flag. It is also used in conjunction with other flags to determine whether or not the adapter can perform vertical parity on transmission.

TERMINATE BLOCK

The routine that performs an INITIATE REQUEST must respond to a previous TERMINATE BLOCK. If the TERMINATE BLOCK never occurs, the logic for the response is omitted.

SYNC = TRUE/FALSE

The NDL programmer is capable of permitting the synchronous characters on a synchronous line to be returned to the MCS. During each receive of a synchronous character, the flag must be checked to determine whether or not the SYNC character is discarded. If the instruction has never been used, the code for this checking and saving of the synchronous character is not generated.

CRC = TRUE/FALSE

CRC is similar to the SYNCHRONOUS flag.

SHIFT = UP/DOWN/MIDDLE

SHIFT is similar to the SYNCHRONOUS flag.

STATION = LIS/VARIABLE

The information pertaining to STATION indicates whether or not the station byte variable has ever been changed. If changed, the microcode must validate the active station during certain operations. The INITIATE REQUEST/INITIATE ENABLE INPUT must either determine that the current station is legal or discontinue the operation. If the station variable is not replaced, then the routines to determine validity are omitted.

Following is an example of the other general information NPC retrieves from the NDLSYS file.

If only one control/request set exists in a code file, the INITIATE/TERMINATE logic uses GOTOS to transfer execution. If more than one set exists, a switch table is used to get to the proper control/request via the logical terminal number.

DEBUG OPTION

For debugging purposes, NPC contains a run-time option (DEBUG) to display information about program execution. Throughout the shell of the post compiler, there are procedure calls for DUMP.BUFFER with

parameters to display certain pieces of data. The parameters consist of: 1) a character string that is displayed with the data; 2) a data identifier; and 3) the number of bytes to be displayed.

Following are standards for displaying certain types of data:

1. Any time a disk record is retrieved, the buffer is displayed along with a character string that indicates the type of data displayed. Record retrieval includes: S-instructions, table data (line, station, translation, etc.), and label references.
2. Each translated S-instruction is displayed.
3. Each generated microinstruction is displayed.

Additionally, several other kinds of data are displayed. Consequently, the DEBUG option generates a listing that shows the flow of execution and displays the microcode in the order generated.

Printer File Listing

Figure 13-3 is a sample PBM file listing. Following the figure is a description of each field.

NPC extracts the following printer file information from M.CODE.FILE:

1. The label field (GET.STO.DELAY in figure 13-3.)
2. The field that specifies the label where a CALL or GOTO must pass control (GET.STO.DELAY in the CALL statement shown in figure 13-1).
3. The actual microcode (hexadecimal value) retrieved from the code file created.

NPC then generates (from the extracted information) the remaining listing requirements, that is, microcode mnemonics and microaddresses. The information pertaining to the S-instructions (S-code and S-address) is retrieved from the NDLSYS file.

1	2	3	4	5	6	7	8	9
NOOP	0000	004A						
ATOGF	0001	0104						
	0002	1C08	INPUT.REQUEST.00	M1 <-- K M1 <-- M1 + B0 <-- N1 B0 <-- B0 & I1 <-- B0	STN.TOGGLE TOGGLE MASK FALSE	%STATION TABLE BASE %APPLY TOGGLE MASK %RESTORE THE BYTE	19BF C2 19C0 0A1000 19C3 7B 19C4 15FE 19C6 89	
ITXMT	0003	0020		B1 <-- B1 & CALL J +	ZERO GET.STD.DELAY.00	%INITIALIZE B1	19C7 0F00 19C9 1E4017	
XTLIT	0004	045C		M1 <-- L + B0 <-- N1 B0 <-- B0 & I1 <-- B0 B0 <--	LNE.FLAGS.4 RESET.IGNORE.BREAK EOT PRE.MANAGER.XMT.B.0 S.LABEL 0084		19CC 134300 19CF 7B 19D0 15EF 19D2 89 19D3 3A04 19D5 1E0206 19D8 10801C	
	0005	R0084		CALL J + IF 7S GOTO J +				
XTRAD	0006	0050		M1 <-- L + B0 <-- N1 B0 <-- B0 & I1 <-- B0 M1 <-- K M1 <-- M1 + B0 <-- I1	LNE.FLAGS.4 RESET.IGNORE.BREAK STN.RECV.AD.1	%SIN TABLE BASE	19DB 134300 19DE 7B 19DF 15EF 19E1 89 19E2 C2 19E3 0A0000 19E6 64 19E7 1E0206 19EA 10801C 19ED C2 19EE 0A0000 19F1 64 19F2 1E0206 19F5 10801C	
	0007	R0084		CALL J + IF 7S GOTO J + M1 <-- K M1 <-- M1 + B0 <-- I1 CALL J + IF 7S GOTO J +	PRE.MANAGER.XMT.B.0 S.LABEL 0084 STN.RECV.AD.2 PRE.MANAGER.XMT.B.0 S.LABEL 0084			
XTLIT	0008	705C		M1 <-- L + B0 <-- N1 B0 <-- B0 & I1 <-- B0 B0 <--	LNE.FLAGS.4 RESET.IGNORE.BREAK L(P) PRE.MANAGER.XMT.B.0 S.LABEL 0084		19F8 134300 19FB 7B 19FC 15EF 19FE 89 19FF 3A70 1A01 1E0206 1A04 10801C	
	0009	R0084		CALL J + IF 7S GOTO J +				
XTLIT	000A	055C		M1 <-- L + B0 <-- N1 B0 <-- B0 & I1 <-- B0 B0 <--	LNE.FLAGS.4 RESET.IGNORE.BREAK ENO PRE.MANAGER.XMT.B.0 S.LABEL 0084		1A07 134300 1A0A 7B 1A0B 15EF 1A0D 89 1A0E 3A05 1A10 1E0206 1A13 10801C	
	000B	R0084		CALL J + IF 7S GOTO J +				
FNXMT	000C	000C						

Figure 13-3. PBM File Listing

Field	Meaning
1	This field contains the mnemonic for an NDL source statement. IFTOG represents the "IF TOG[0] THEN" sequence.
2	This field contains a hexadecimal value for the S-level address of the NDL instruction.
3	This field is the actual hexadecimal value of the S-instruction. It is six bytes in length. An "R" in the left margin of this field indicates that the fifth and sixth bytes are the relative S-address of the false branch for the IF Statement.
4	This field contains a label if one is associated with this instruction.
5	This field represents the mnemonic for the microinstruction .
6	Literals or addresses referenced.
7	This field contains any comment that might clarify the actions taking place. Comments are preceded by a percent (%) sign.
8	This field contains the hexadecimal address of the microinstruction.
9	This field contains the hexadecimal value of the instruction. The length of these instructions varies as does the increment of the microaddress. The first byte of the instruction is always the Op code. This field may also include strings of data values that were stored during code generation.

NOTE

Two-byte addresses and literals appear byte-reversed because this is how they are stored in DCP memory.

APPENDIX A

NDL RESERVED WORD LIST

<p>A</p> <p>ABORT ACU ADDERR ADDRESS ALL ALLCASE ANSWERTONE ASC67 ASYNC AUX AUXILIARY</p> <p>B</p> <p>BACKSPACE BAUDOT BCC BCCERR BCL BDI BEGIN BINARY BITS BLOCK BLOCKED BREAK BUFFER BUFFERCOUNT BUFOVFL BUSY BYTE</p> <p>C</p> <p>CARRIAGE CHAR CHARACTER CLEAR CODE CODEFILE CONSTANT CONTINUE CONTROL CONTROLFLAG CRC CRCERR</p>	<p>D</p> <p>DATASET DCP DEFAULT DEFAULTCHAR DELAY DELIVER DIALIN DIALOUT DIRECT DISABLE DISCONNECT DOWN DUPLEX</p> <p>E</p> <p>EBCDIC ECMA ELSE ENABLE ENABLED ENABLEINPUT END ENDOFBUFFER EQ EQL ERROR EVEN EVENT1</p> <p>F</p> <p>FALSE FAMILY FETCH FILE FINISH FLAGFILL FORK FORMATERR FRAMEADDR FREQUENCY</p> <p>G</p> <p>GE</p>	<p>GEQ GETSPACE GLOBAL GO GT GRT</p> <p>H</p> <p>HIGH HOME HORIZONTAL</p> <p>I</p> <p>IDLE IF IIR INCREMENT INITIALIZE INITIATE INPUT INVALID IR</p> <p>L</p> <p>LCHAR LE LEQ LIBRARY LIMIT LINE LINEDELETE LINEFEED LOGIN LOSSOFCARRIER LOW LOWER LS LSS</p> <p>M</p> <p>MAXINPUT MAXSTATIONS MEMORY</p>	<p>MICRO MIDDLE MILLI MIN MOD MODEM MYUSE</p> <p>N</p> <p>NAKFLAG NAKONSELECT NE NEQ NOINPUT NOISEDELAY NORMAL NOSPACE NOT NULL</p> <p>O</p> <p>ODD OUTPUT</p> <p>P</p> <p>PAGE PAPERMOTION PARITY PAUSE PHONE PRIORITYSAVE</p> <p>Q</p> <p>QUEUED</p> <p>R</p> <p>RATESELECT READY RECEIVE REQUEST RESTORE RETRY RETURN</p>
--	---	---	---

Appendix A
NDL Reserved Word List

S	
SAVE	TRAN
SCREEN	TRANERR
SEC	TRANSLATE
SHIFT	TRANSLATION
SKIP	TRANSMISSION
SKIPCONTROL	TRANSMIT
SPACE	TRANSMITDELAY
SPEED	TRANSPARENT
SPO	TRUE
STANDBY	TURNAROUND
STATION	TYPE
STOPBIT	U
STORE	UNDERFLO
SUM	UP
SYNC	UPPER
SYNCS	V
T	VALID
TALLIES	VERTICAL
TALLY	W
TELEX	WAIT
TERMINAL	WIDTH
TERMINATE	WRAPAROUND
TEXT	WRU
THEN	WRUFLAG
TIMEOUT	X
TIMER	XMTADD
TO	
TOG	
TOGS	

APPENDIX B

NETWORK SEMANTIC CONSIDERATIONS

Although an NDL program is highly modularized, there are many areas where the interaction of several different definitions is important. The following is a list of conditions which affect these areas. They are organized according to section to section interfaces.

TERMINAL - CONTROL/REQUEST

TERMINATE ENABLEINPUT

This instruction cannot appear in receive requests.

TRANSMIT TEXT

This instruction cannot appear in receive requests.

RECEIVE TEXT

This instruction cannot appear in transmit requests.

GETSPACE

This instruction cannot appear in transmit requests.

STORE

STORE CHARACTER or STORE STRING cannot appear in transmit requests.

TEXT CONTROL CHARACTERS

The following characters must be defined for any terminal which uses a control or request set which references them:

- END
- BACKSPACE
- LINEDDELETE
- WRU

WRUFLAG

The WRU character must be defined for any terminal which uses a request set which references WRUFLAG.

STATION - CONTROL/REQUEST

CONTROL

The control character must be defined for any station which uses a control or request set which references CONTROL.

CONTROL FLAG

The control character must be defined for any station which uses a request set which references CONTROL FLAG.

STATION - MODEM

TYPE

A station and its modem must specify the same type (async/sync).

SPEED

The speed of a sync station must be equal to the speed of its modem. For async stations, the speed must be less than or equal to that of the modem.

STATION - TERMINAL

TYPE

A station and its terminal must specify the same type (async/sync).

TYPE OPTION

If a station specifies a TYPE option, its terminal must have the capability. If the station gives NO TYPE option, the terminal must specify only one option and the station uses that value. If the WRAPAROUND option is specified, the terminal must not be full-duplex.

SPEED

If a station specifies a SPEED, it must be less than or equal to the terminal speed for asynchronous operation and equal to the terminal speed for synchronous. If no station speed is given, the terminal must specify only one speed and the station uses that value.

STOPBITS

If a station SPEED statement specifies the number of stopbits, the corresponding terminal speed must have the same number of stopbits. If the station does not specify the number of stopbits, the terminal value is used.

MYUSE

A terminal definition must contain a request for each I/O capability specified in the station MYUSE statement.

ADDRESS

A station address must agree in size with the terminal address specification.

WIDTH

The station width must be less than or equal to the terminal width. If no station width is specified, it defaults to the terminal value. If no terminal width is given, a default value of 255 is used.

PAGE

The station page size must be less than or equal to the terminal PAGE size. If no station page size is given, it defaults to the terminal value.

WRAPAROUND

If a station does not have a WRAPAROUND statement, the terminal value is used.

LINE - MODEM

A line modem must have the same TYPE (async/sync) as stations on the line.

SPEED

On sync lines, the line modem speed must be equal to the speed of all stations on that line. For async lines, the station's speed must be less than or equal to that of the modem.

DIALIN

The line modem on a dialin line must have dialin capability.

DIALOUT

The line modem on a dialout line must have dialout capability.

DUPLEX

All modems on a duplex line, including station modems, must have duplex capability.

LINE - TERMINAL

DUPLEX

A duplex terminal may only be used on a duplex line.

CONSISTENCY ALONG LINE

The following characteristics must be identical for all terminals on a line:

- BYTE SIZE (including parity bit)
- CONTROL SET
- VERTICAL PARITY CHECK
- VERTICAL PARITY TYPE (odd/even)

LINE - STATION

CONNECT TYPE

The stations on a line must have the same connect type (DIRECT/TELEX/BDI/MODEM) as the line.

CONSISTENCY ALONG LINE

The following characteristics must be identical for all stations on a line:

- TYPE (async/sync)
- TYPE OPTION
- SPEED
- NUMBER OF STOPBITS
- EXISTENCE OF STATION ADDRESSES

APPENDIX C SAMPLE NDL PROGRAM CONSTANT SECTION

CONSTANT

NULL =	4"00".	
SOH =	4"01".	%START OF HEADER
STX =	4"02".	%START OF TEXT
ETX =	4"03".	%END OF TEXT
EOT =	4"04".	%END OF TRANSMISSION
ENQ =	4"05".	%ENQUIRE
NAK =	4"15".	%NEGATIVE ACKNOWLEDGE
FSL =	4"73".	

LINE CONTROL SECTION

LINE (TALLY[0]) – Reflects the number of times that line control was entered during the last station cycle (i.e. STATION running between MAXSTATIONS-1 and 0). Line control is entered at the top when a request set is terminated. Possible values for LINE (TALLY[0]) range between 0 and MAXSTATIONS-1.

LINE(TALLY[1]) – Reflects the value of LINE(TALLY[0]) at the end of the last cycle, unless LINE(TALLY[0]) = 0. In this case, LINE(TALLY[1]) = 1. LINE(TALLY[1]) is used in conjunction with polling frequency.

CONTROL POLL:

```

% * * * * *
LINE(TALLY[0]) = LINE(TALLY[0]) + 1.
LINE(QUEUED) = TRUE.
0:
IF STATION >0 THEN
BEGIN
1:
    PAUSE.
    STATION = STATION-1.
    INITIATE REQUEST.
%     IF STATION IS NOT VALID, READY, OR QUEUED THE NEXT
%     INSTRUCTION WILL BE EXECUTED. IF THE ABOVE CONDITIONS
%     ARE TRUE, THEN THE REQUEST WILL BE ACTIONED AND LINE
%     CONTROL WILL BE RE-ENTERED AT THE TOP.
IF STATION(ENABLED) THEN
    BEGIN
        IF STATION(TALLY) GT LINE(TALLY[1]) THEN
            BEGIN
                PAUSE.
                STATION(TALLY) = STATION(TALLY) -
                    LINE (TALLY[1]).
                % THE LARGER THE VALUE OF LINE(TALLY[1])
                % THE QUICKER THE VALUE OF STATION(TALLY)
                % WILL BE DECREMENTED. THIS WILL AVOID
                % UNDESIRABLE DELAYS, IN POLLING, AFTER
                % PERIODS OF HIGH STATION ACTIVITY.
                LINE(QUEUED) = TRUE.
            END
        ELSE

```

```

                BEGIN
                PAUSE.
                STATION(TALLY) = STATION(FREQUENCY).
                INITIATE ENABLEINPUT.
                END.
        END.

        GO TO 0.
        END.
    PAUSE.
    STATION = MAXSTATIONS.
    IF LINE (QUEUED) THEN % LINE (QUEUED) WILL BE FALSE ONLY IF NO
                        % STATION ON THE LINE IS ENABLED.
                        % IF THIS IS THE CASE, THERE IS NO POINT
                        % IN ATTEMPTING TO POLL A STATION. THEREFORE,
                        % LINE CONTROL ENTERS THE IDLE STATE
                        % TO BE REINITIATED BY WHEN AN OUTPUT
                        % FUNCTION IS QUEUED.

    BEGIN
    LINE (QUEUED) = FALSE.
    IF LINE(TALLY[0]) = 0 THEN % THERE HAS BEEN NO STATION ACTIVITY
        BEGIN
            LINE(TALLY[1]) = 1.
            LINE (BUSY) = FALSE.
            DELAY(1 SEC). % IF AN OUTPUT FUNCTION IS QUEUED
                        % DURING THIS DELAY LINE CONTROL
                        % WILL BE ENTERED AT THE TOP.

            LINE(BUSY) = TRUE.
        END
    ELSE
        BEGIN
            PAUSE.
            LINE(TALLY[1]) = LINE(TALLY[0]).
            LINE(TALLY[0]) = 0.
        END.
    GO TO 1.
    END.
    IDLE.
    REQUEST POLLIT:
    %           ERROR[1] = TIMEOUT:5,
                    STOPBIT:3,
                    BUFOVFL:3,
                    PARITY:4,
                    LOSSOFCARRIER:5.

    %
                    ERROR[2] = TIMEOUT:2,
                    STOPBIT:2,
                    BUFOVFL:2,
                    PARITY:2,
                    LOSSOFCARRIER:2.

    %
    % * * * * *
    %

```

```
TOG[0] = FALSE.
INITIATE TRANSMIT.
TRANSMIT EOT[BREAK:7].
TRANSMIT ADDRESS[BREAK:7].
TRANSMIT POL[BREAK:7].
TRANSMIT ENQ[BREAK:7].
FINISH TRANSMIT.

%
INITIATE RECEIVE.
RECEIVE(1 SEC) [1].
IF CHAR = SOH
  THEN BEGIN
    CONTROLFLAG = FALSE.
    INITIALIZE BCC.
    RECEIVE(1 SEC) ADDRESS [1,ADDRESS:3].
    GETSPACE[6].
    RECEIVE TRAN[1, TRANERR:NULL].
    RECEIVE STX[1, FORMATERR:3].
    INITIALIZE TEXT.
    RECEIVE[1,ETX:1,CONTROL].
    STORE[ENDOFBUFFER:3].
    RECEIVE TEXT [1,ETX,ENDOFBUFFER:3].
    RECEIVE BCC[1,BCCERR:3].

1:
%
    INITIATE TRANSMIT.
    TRANSMIT ACK.
    FINISH TRANSMIT.

%
    INITIATE RECEIVE.
    RECEIVE(1 SEC) EOT [2, FORMATERR:NULL].
2:
    INCREMENT TRAN.
    TERMINATE NORMAL.
  END.
  IF CHAR = EOT THEN TERMINATE NOINPUT.

%
3:
  RECEIVE(25 MILLI) [1].
  GO TO 3.
4:
  IF CHAR NEQ 4"F" THEN GO TO 3.
5:
  IF TOG[0] THEN TERMINATE NOINPUT.
  IF RETRY = 0 THEN TERMINATE ERROR.
  RETRY = RETRY - 1.
  TERMINATE NOINPUT.
6:
  STATION(TALLY) = 0.
  TOG[0] = TRUE.
  GO TO 3.
7:
  FINISH TRANSMIT.
  INITIATE RECEIVE.
  GO TO 3.
```

REQUEST SELECTIT:

```
%
%
```

Appendix C
Sample NDL Program

```
ERROR[1] = TIMEOUT:4,  
          STOPBIT:2,  
          BUFOVFL:2,  
          PARITY:3,  
          LOSSOFCARRIER:4.
```

```
%  
% * * * * *  
%
```

```
TOG[0] = FALSE.  
INITIATE TRANSMIT.  
TRANSMIT EOT[BREAK:5].  
TRANSMIT ADDRESS[BREAK 5].  
TRANSMIT SEL[BREAK:5].  
TRANSMIT ENQ[BREAK:5].  
FINISH TRANSMIT.
```

```
%
```

```
INITIATE RECEIVE.  
RECEIVE(1 SEC) [1].  
IF CHAR = ACK  
  THEN BEGIN  
    INITIATE TRANSMIT.  
    TRANSMIT SOH[BREAK:5].  
    INITIALIZE BCC.  
    TRANSMIT ADDRESS[BREAK:5].  
    TRANSMIT TRAN[BREAK:5].  
    TRANSMIT STX[BREAK:5].  
    INITIALIZE TEXT.  
    TRANSMIT TEXT[BREAK:5].  
    TRANSMIT ETX[BREAK:5].  
    TRANSMIT BCC[BREAK:5].  
    FINISH TRANSMIT
```

```
%
```

```
INITIATE RECEIVE.  
RECEIVE(1 SEC) [1].  
IF CHAR = ACK.  
  THEN BEGIN  
    INCREMENT TRAN.  
    TERMINATE NORMAL.  
  END.  
IF CHAR NEQ ACK THEN GO TO 2.  
NAK FLAG = TRUE.  
GO TO 4.  
END.  
IF CHAR = NAK  
  THEN BEGIN  
    NAKONSELECT = TRUE.  
    TOG[0] = TRUE.  
    GO TO 4.  
  END.
```

- 2: RECEIVE(25 MILLI) [1].
GO TO 2.
- 3: IF CHAR NEQ 4"FF" THEN GO TO 2.

```
4:  IF RETRY = 0 THEN TERMINATE ERROR.  
    RETRY = RETRY - 1.  
    IF TOG[0] THEN TERMINATE ENABLEINPUT.  
    TERMINATE NOINPUT.  
% * * * * *  
%  
5:  FINISH TRANSMIT.  
    INITIATE RECEIVE.  
    GO TO 2.
```

MODEM SECTION

```
MODEM M202D:                                     %VALUES FOR LEASED  
    TRANSMITDELAY = 200 MILLI.                   % LINES  
    NOISEDELAY = 0.  
    TYPE = ASYNC.  
    SPEED = 1200.  
MODEM M202C:                                     %VALUES FOR SWITCHED  
    TRANSMITDELAY = 175 MILLI.                   % LINES  
    NOISEDELAY = 120 MILLI.  
    TYPE = ASYNC. DIALIN.  
    SPEED = 1200.  
MODEM SUPER:  
    TRANSMITDELAY = 36 MILLI.  
    NOISEDELAY = 0.  
    TYPE = ASYNC.  
    SPEED = 9600.
```

TERMINAL SECTION

TERMINAL DEFAULT SGTEST:

ADDRESS = 2.
SPEED = 1200, 1800, 2400.
TURNAROUND = 0.
TIMEOUT = 1 SEC.
CODE = ASC67.
PARITY - VERTICAL: EVEN, HORIZONTAL: EVEN.
CONTROL = POLL.
REQUEST = POLLIT: RECEIVE, SELECTIT: TRANSMIT.
MAXINPUT = 255.
BLOCKED = FALSE.
END = ETX.
BACKSPACE = BS.
LINEDELETE = DEL.
WRU = ENQ.
TYPE = ASYNC(MODEM).
BYTE = 7, PARITY.

TERMINAL TC500:

DEFAULT = SGTEST.
TRANSMISSION = 3.
SPEED = 1200.

TERMINAL TC3500MA:

DEFAULT = SGTEST.
TRANSMISSION = 3.
SPEED = 9600.

STATION SECTION

STATION DEFAULT DEFAULTSTATION:

CONTROL = 4"40.
ENABLEINPUT = TRUE.
FREQUENCY = 0.
LOGIN = TRUE.
MYUSE = INPUT, OUTPUT.
RETRY = 10.
WIDTH = 72.
WRAPAROUND = TRUE.

STATION STATION4:

DEFAULT = DEFAULTSTATION.
MODEM = M202D.
SPEED = 1200.
ADDRESS = "A1".
TERMINAL = TC500.
TYPE = ASYNC(MODEM).

STATION STATION5:

DEFAULT = DEFAULTSTATION.
MODEM = SUPER.
SPEED = 9600.
ADDRESS = "A1".
TERMINAL = TC3500MA.
TYPE = ASYNC(MODEM).

STATION STATION6:

DEFAULT = DEFAULTSTATION.

SPEED = 1200.
ADDRESS = "A1".
TERMINAL = TC500.
TYPE = ASYNC(MODEM).
MODEM = M202C.

LINE SECTION

LINE LINE4:
ADDRESS = 0:3.
MAXSTATIONS = 2.
TYPE = MODEM.
MODEM = M202D.
STATION = STATION4.

LINE LINE5:
ADDRESS = 0:4.
MAXSTATIONS = 1.
TYPE = MODEM.
MODEM = SUPER.
STATION = STATION5.

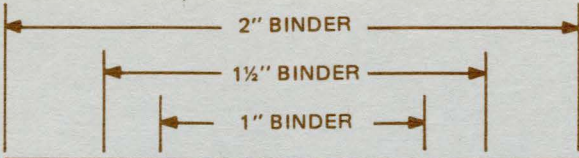
LINE LINE6:
ADDRESS = 0:5.
MAXSTATIONS = 2.
TYPE = MODEM, DIALIN.
MODEM = M202C.
STATION = STATION6.

DCP SECTION

DCP 0:
MEMORY = 48000.
BUFFER = 256.
BUFFERCOUNT = 10.

FILE SECTION

FILE FILE1:
FAMILY = STATION4, STATION5.
FILE FILE2:
FAMILY = FILE1, STATION6.



Computer Mgmt. System (CMS)
Network Definition Language (NDL)
REFERENCE MANUAL

1090925
Printed in U.S.A.