NIAKWA PROGRAMMING LANGUAGE

TECHNICAL REFERENCE GUIDE

STATEMENTS GUIDE

1st Edition - July 1993
COPYRIGHT © 1993 Niakwa, Inc.

Niakwa, Inc.
23600 N. Milwaukee Avenue
Mundelein, IL 60060

PHONE: (708) 634-8700 FAX: (708) 634-8718 TELEX: 3719965 NIAK UB

DISCLAIMER OF WARRANTIES AND LIMITATION OF LIABILITIES AND PROPRIETARY RIGHTS

The staff of Niakwa, Inc. (Niakwa) has taken due care in preparing this manual. Nothing contained herein shall be construed to modify or alter in any way the standard terms and conditions of the Niakwa Programming Language (NPL) Support and Distribution License Agreement, the End-User Support Only License Agreement, the Niakwa Software License Agreement and Warranty, or any other Niakwa License Agreement (collectively, the "License Agreements") by which this software package was acquired.

This manual is to serve as a guide for use of the Niakwa software only and not as a source of representations or additional undertakings by Niakwa. The licensee must refer to the License Agreements for Niakwa product and service representations.

No ownership of Niakwa software is transferred by any of the License Agreements. Any use of Niakwa software beyond the terms and conditions of the License Agreements, without the written authorization of Niakwa, is prohibited.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without prior written permission from Niakwa, Inc.

Niakwa is a registered trademark of Niakwa Management Services 1975 Ltd. and is licensed to Bluebird Systems.

Niakwa Programming Language (NPL), Bluebird, and SuperDOS are registered trademarks of Bluebird Systems.

All other trademarks are property of their respective holders.

PREFACE

The Niakwa Programming Language (NPL) Technical Reference Guide consists of two manuals: the Programmer's Guide and the Statements Guide. The Technical Reference Guide is intended as a hardware-independent reference for programmers in the correct use of the NPL and its program development and debugging facilities. It should be used in conjunction with the NPL Supplements, which provide operating system-specific information as it relates to NPL and its installation, and the Runtime User's Guide, which provide platform-specific information on installing and operating the NPL Runtime.

NOTE: Refer to the Programmer's Guide Preface for a complete overview of the NPL documentation.

Table of Contents

PREFACE

INTRODUCTION

Overview 1-1
Notational Conventions 1-1
Form of Presentation 1-2
Statement Description Layout 1-3
Statement Conventions 1-4
Terminology 1-7
Variable Names 1-9
Organization of the Statements Guide 1-14

LANGUAGE STATEMENTS

Overview 2-1
ABS Function 2-2
ADD[C] Alpha-operator 2-3
+=numeric expression 2-5
$ALERT 2-6
ALL Alpha-operand 2-7
AND Alpha-operator 2-9
ARC COS Function 2-11
ARC SIN Function 2-12
ATN Function - ARC TANGENT 2-13
BIN Function/Alpha-operand 2-14
BOOL Alpha-operator 2-16
$BOXTABLE 2-19
BREAK 2-23
$BREAK 2-25
CASE Default 2-26
CASE Logical 2-28
CASE Numeric 2-30
CASE String 2-32

CLEAR 2-35
$CLOSE 2-38
COM 2-40
COM CLEAR 2-44
& (Concatenation) Alpha-Operator........ 2-46
CONTINUE........ 2-47
CONTINUE LOAD 2-48
CONTINUE NEXT........ 2-49
CONTINUE RETURN........ 2-51
CONVERT 2-55
COPY........ 2-59
COS Function 2-63
DAC Alpha-operator 2-60
DATA 2-62
DATA LOAD BA 2-63
DATA LOAD BM........ 2-65
DATA LOAD DA 2-68
DATA LOAD DC 2-71
DATA LOAD DC OPEN........ 2-73
DATA SAVE BA 2-76
DATA SAVE BM 2-80
DATA SAVE DA........ 2-85
DATA SAVE DC........ 2-88
DATA SAVE DC CLOSE 2-90
DATA SAVE DC OPEN 2-91
DATE........ 2-94
DBACKSPACE........ 2-96
DEFFN' Keyboard Input........ 2-98
DEFFN' Subroutine........ 2-100
DEFFN Function Definition........ 2-104
DEFFN@PART........ 2-106
DELETE 2-108
$DEMO 2-110
$DET........ 2-113
$DET (cont.)........ 2-114
$DEVICE........ 2-115
DIM........ 2-126
DIM Constant Variable Declarations........ 2-131
DIM /PUBLIC 2-133
DIM /RECURSIVE........ 2-135

DIM /STATIC 2-137
$DISCONNECT 2-140
DO/ENDDO 2-141
DSC Alpha-operator 2-146
DSKIP 2-148
ELSE 2-146
ELSE Structured 2-148
END 2-149
$END 2-150
END FUNCTION 2-152
END IF 2-153
END PROCEDURE 2-154
END PUBLIC 2-155
END RECORD 2-156
END SWITCH 2-157
ERR Function 2-158
ERR$ 2-160
ERROR 2-162
EXEC Key 2-165
EXP Function 2-166
FIELD 2-167
String FIELD-Expressions - Alpha-Variable Equivalent 2-170
Numeric FIELD-Expressions - Term in Numeric Expression 2-172
$FIELDFORMAT Function 2-173
#FIELDLENGTH Function 2-175
#FIELDSTART Function 2-177
FIX Function 2-179
FN Function 2-180
FOR/BEGIN Structured 2-182
FOR/TO 2-184
$FORMAT DISK 2-187
FUNCTION 2-188

’Function-name (...) Numeric-Expression Equivalent 2-191
’Function-name$(...) Literal-String Equivalent 2-192
$GIO 2-193
#GOLDKEY Function 2-200
GOSUB 2-203
GOSUB’ 2-205
GOTO 2-209
HALT Key 2-211

$HELP 2-212
$HELPINDEX 2-214
HEX Function 2-217
HEXPACK 2-219
HEXPRINT 2-220
HEXUNPACK 2-221
#ID Function 2-222
$IF 2-223
IF/THEN 2-228
IF END THEN 2-234
IMAGE (%) 2-236
INCLUDE 2-238
INIT 2-241
INPUT 2-242
INPUT SCREEN 2-245
INT function 2-254
$KEEPREMS 2-255
$KEYBOARD 2-259
KEYIN 2-264
LEN Function 2-267
LET Alpha Assignment 2-269
LET Numeric Assignment 2-273
LET Numeric Field Assignment 2-277
LGT Function 2-278
LIMITS 2-279
LIMITS INDEX 2-282
LINPUT 2-284
LIST (General Parameters) 2-288
LIST 2-290
LIST # 2-295
LIST ' 2-299
LIST DC 2-304
LIST DIM 2-309
LIST DT 2-314
LIST FIELD 2-318
LIST FUNCTION 2-322
LIST PROCEDURE 2-326
LIST PUBLIC DEFFN 2-330
LIST PUBLIC FIELD 2-334
LIST PUBLIC FUNCTION 2-338

LIST PUBLIC PROCEDURE 2-342
LIST PUBLIC RECORD 2-345
LIST PUBLIC V 2-348
LIST RECORD 2-351
LIST Statement Label References 2-355
LIST STACK 2-359
LIST STACK DIM 2-363
LIST T 2-367
LIST V 2-371
LOAD Command 2-376
LOAD Statement 2-377
LOAD’ 2-380
LOAD BOOT Command 2-382
LOAD DA Command 2-384
LOAD DA Statement 2-386
LOAD RUN 2-389
LOG Function 2-391
LOOP 2-392
$MACHINE 2-394
MAT CON 2-400
MAT COPY 2-401
MAT IDN 2-404
MAT INPUT 2-405
MAT INV 2-407
MAT MERGE 2-410
MAT MOVE 2-416
MAT* (Multiply) 2-423
MAT PRINT 2-424
MAT READ 2-425
MAT REDIM 2-426
MAT SEARCH 2-429
MAT SORT 2-434
MAT TRN 2-441
MAT ZER 2-442
MAT Addition 2-443
MAT Assignment 2-444
MAT Scalar Multiplication 2-445
MAT Subtraction 2-446
MAX Function 2-448
MIN Function 2-449

MOD Function 2-450
MODULE Command 2-451
MOVE 2-452
MOVE END 2-455
$MSG 2-457
$NAMEOF() - Built-in String Function 2-456
$NETID 2-458
NEXT 2-459
NEXT CLEAR 2-461
NUM Function 2-463
$NUMBERS 2-464
$OBJECT 2-466
ON ERROR 2-468
ON/GOSUB 2-471
ON/GOTO 2-473
ON/SELECT 2-475
$OPEN 2-477
$OPTIONS 2-479
OR Alpha-operator 2-491
$OSERR 2-493
PACK 2-495
$PACK 2-497
#PART Function 2-521
#PI Function 2-522
POS Function 2-523
PRINT 2-525
PRINT AT Function 2-529
PRINT BOX Function 2-531
PRINT HEXOF Function 2-533
PRINT SCREEN 2-534
PRINT TAB Function 2-538
PRINT TO 2-539
$PRINTER 2-542
PRINTUSING 2-544
PRINTUSING TO 2-548
PROCEDURE 2-550
'Procedure-name (Call PROCEDURE) 2-553
$PROGRAM 2-554
$PSTAT 2-556
PUBLIC 2-558

READ 2-560
READ DC 2-563
RECORD 2-566
#RECORDLENGTH Function 2-568
$RELEASE PART 2-569
$RELEASE TERMINAL 2-570
REM 2-572
REM $PC 2-574
RENAME 2-576
RENAME DEFFN' 2-577
RENAME FIELD 2-579
RENAME FUNCTION 2-581
RENAME PROCEDURE 2-583
RENAME RECORD 2-585
RENAME = (Statement Label) 2-588
RENAME V 2-590
RENUMBER 2-592
REPEAT 2-595
RESAVE 2-596
RESET 2-598
RESTORE 2-600
RETURN 2-602
RETURN CLEAR 2-605
$REV 2-607
RND Function 2-609
ROTATE 2-610
ROUND Function 2-612
RUN Command 2-613
RUN Statement 2-615
SAVE 2-618
SAVE BOOT Command 2-623
SAVE DA 2-625
SCRATCH 2-628
SCRATCH DISK 2-630
$SCREEN 2-633
SELECT 2-635
$SELECT 2-637
SELECT @PART 2-640
SELECT CI 2-641
SELECT CO 2-642

SELECT D,R,G 2-644
SELECT DISK/FILE-NUMBER 2-646
SELECT DRIVER 2-648
SELECT ERROR 2-649
SELECT INPUT 2-652
SELECT LINE 2-653
SELECT LIST 2-649
SELECT LISTLINE 2-650
SELECT LOG 2-651
SELECT ON/OFF 2-654
SELECT ON CLEAR 2-655
SELECT P 2-656
SELECT PLOT 2-657
SELECT PRINT 2-658
SELECT ROUND 2-660
SELECT TAPE 2-661
SELECT TC 2-662
SELECT TERMINAL 2-663
$SER 2-665
SET DATA 2-666
SET PROGRAM 2-668
SGN Function 2-671
$SHELL 2-672
SIN Function 2-675
$SOURCE Function 2-676
SPACE Function 2-680
SPACEF Function 2-683
SPACEK Function 2-684
SPACEP Function 2-686
SPACEV Function 2-687
SPACEW Function 2-688
SQR Function 2-689
= statement-name (Statement Labels) 2-690
STEP 2-692
STEP # 2-695
STEP OFF 2-697
STOP 2-698
STR() Function 2-701
SUB[C] Alpha-operator 2-704
-= numeric-expression 2-706

SWITCH Logical....2-707
SWITCH Numeric....2-710
SWITCH String....2-712
$TAB....2-714
TAN Function....2-716
#TERM....2-717
TIME....2-718
TRACE....2-720
TRACE OFF....2-722
TRACE #....2-723
TRACE '....2-726
TRACE V....2-729
$TRAN....2-733
UNPACK....2-735
$UNPACK....2-737
UNSCRATCH....2-751
UNTIL....2-754
USES....2-755
VAL Function....2-757
VER Function....2-760
VERIFY....2-762
WEND....2-764
WHILE....2-765
XOR Alpha-operator....2-766

LIBRARY FUNCTIONS

Overview....3-1
Development Package Files for Library Functions....3-2
Changes to $SOURCE Functionality to Support LIN's....3-3
'SourceioGetTableLengths....3-3
'SourceioLoadIdentifierTable....3-4
'SourceioReadLine....3-4
$SOURCE....3-4
'SourceioCloseObjectFile....3-4
$OBJECT Functionality Changes for LIN's....3-5
'ObjectioCreateFile....3-5
'ObjectioClearIdentifierTable....3-5
'ObjectioAppendLine....3-5
'ObjectioAppendLongIdentifierTable....3-6

’ObjectioCloseFile...3-6
FIELD Type Specifications...3-6
Defining Field Type Using $PACK Mnemonic Codes...............................3-7
FIELD Type NDM Specifications..3-8
Defining Field Type Using NDM Mnemonic Codes 3-8

APPENDIX A

RESERVED WORDS TABLE

LANGUAGE COMPATIBILITY CHART

Overview...B-1

CHAPTER 1

INTRODUCTION

1.1 Overview

This chapter provides general information relating to this Statements Guide and its use.

Section 1.2 explains the notational conventions used in this guide.

Section 1.3 discusses the organization and presentation of material in this Statements Guide.

1.2 Notational Conventions

The NPL Statements Guide uses the following notational conventions.

NOTE: Notes provide information of particular importance.

WARNING--Warnings are special conditions that require extra care by the user.

Hint: Hints provide helpful comments pertaining to the use of particular features.

1.2.1 Form of Presentation

The complete set of all NPL instructions is found in Chapter 2 of this guide, in alphabetical order. If an instruction begins with a special character ($, =,or #), the special character is ignored for ordering purposes. For example, the instruction #ID is located between HEXUNPACK and IF.

For each instruction, the verb definition, type, and English equivalent are shown.

In the verb type area, the NPL instructions are classed as either statements, functions, or operators. A statement is a programmable instruction. A function is used to construct numeric or alpha-expressions within a statement. Operators perform operations on alpha-operands and can be used only within alpha-expressions on the right side of an alpha LET statement.

Verbs which are used as functions or operators are so indicated on the instruction definition line.

In the "Compatibility Issues" area of each statement, the following issues are discussed:

1. Compatibility with earlier revisions of NPL

   If a statement is not supported on all NPL revisions (starting with 1.00.02), the revision number and release date upon which it was first supported are specified.

2. Compatibility with Wang 2200 Basic-2

   All differences in syntax or functionality from the Wang 2200 Basic-2 language are described.

3. NPL platform versions

   NPL operates on many different computers under many different operating systems. Although every effort is made to achieve complete compatibility across different platforms, operating systems and environments, 100% compatibility is not possible. Incompatibilities based on different hardware versions which do result are primarily limited to certain I/O routines. Any statement which has even potential incompatibilities across platform versions is so indicated. In all cases where incompatibilities are noted in this section, the incompatibilities are fully described in the appropriate NPL operating system-specific Supplement.

1.2.2 Statement Description Layout

Information pertaining to each statement is delineated in a group of specific categories. These categories are:

- General Form. This section shows the statement format, along with variables, operators or other parameters. The general form of the statement is enclosed within a syntax box, and attributes of variables or operators are defined within this box. Refer to the next section for conventions used to display the general form of NPL statements.

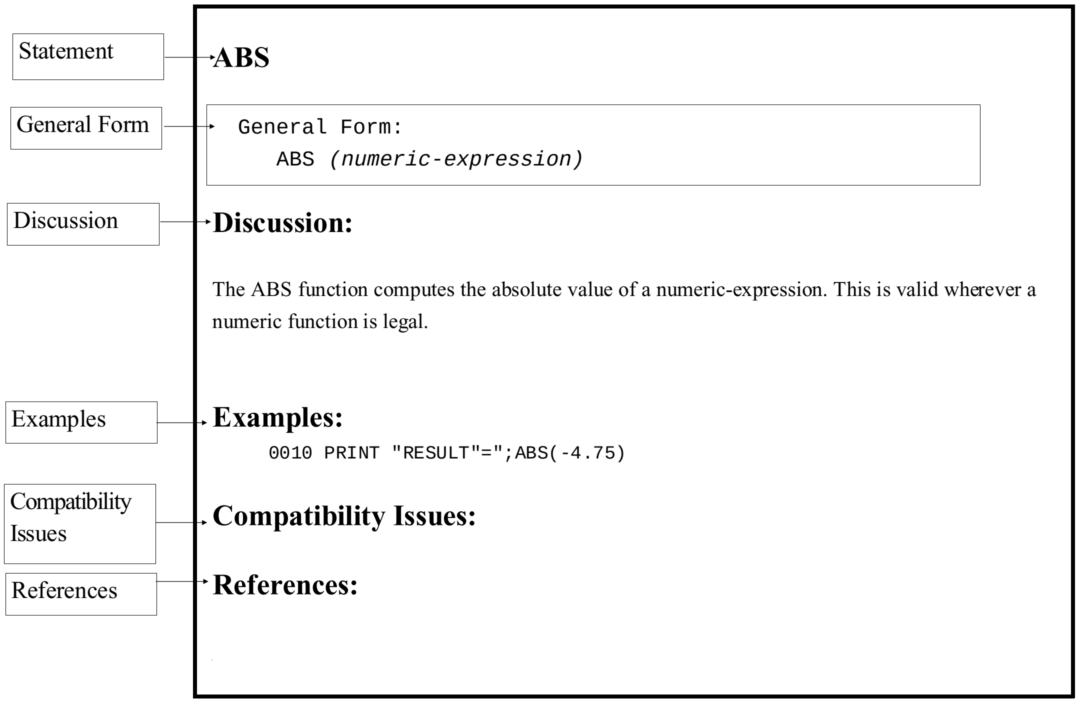
- Discussion. This section explains the purpose and circumstances for using the statement.

- Examples. This section provides one or more examples of how the statement may be used in writing programs.

- Compatibility Issues. This section describes any considerations or problems with using this statement in conjunction with Wang Basic-2 programs.

- References. This section lists any statements related to the statement being described.

Here is an illustration of the statement description layout.



ABS

```
General Form:
   ABS (numeric-expression)
```

Discussion:

The ABS function computes the absolute value of a numeric-expression. This is valid wherever a numeric function is legal.

Examples:

```
0010 PRINT "RESULT=";ABS(-4.75)
```

Compatibility Issues:

References:

Illustration of Statement Description

1.2.3 Statement Conventions

Format conventions are used to illustrate the various elements of NPL statements. These conventions are described below.

- Each statement appears on a separate page, with the statement as a page header.
- The general form of each statement is enclosed within a box.
- Uppercase letters ("A" through "Z"), digits ("0" to "9"), and special characters (such as "$", "#", ":") must always appear exactly as presented in the general format.
- All lower-case words indicate information that the user must supply. These words appear in *italic* type.

For example:

```
LEN (alpha-variable)
```

The user must supply the alpha-variable.

- When braces, "{ }", enclose a vertically stacked list, or a horizontal list with each item separated by a comma (","), the user must choose one of the options within braces. Information within braces is shown in *italic* type.

  For example:

  ```
  ALL ({literal-string, alpha-variable, two-hexdigits})
  ```

  or

  ```
  ALL   ({literal-string  })
         {alpha-variable  }
         {two-hexdigits   }
  ```

Here, the ALL instruction must be followed by one and only one of the items in the list.

- Brackets, "[ ]", indicate that the enclosed items are optional. When brackets enclose a vertical list or a horizontal list, the user may specify one or none of the items. Information within brackets is shown in *italic* type.

  For example:

  ```
  INPUT [literal-string,] variable [,variable]...
  ```

  Here, the INPUT instruction may optionally contain a literal-string followed by an optional comma preceding the required "variable". Additional variables may optionally be specified.

  or:

  ```
  CLEAR [V                                ]
        [N                                ]
        [P [line-number1][,[line-number2]]]
  ```

Here, either the V, N, or P parameter may be specified, but no parameter is required.

NOTE: Here, line-number parameters may be optionally specified only if the "P" parameter is specified.

- The presence of an ellipsis (...) within any format indicates that the unit immediately preceding the ellipsis can occur one or more times in succession.

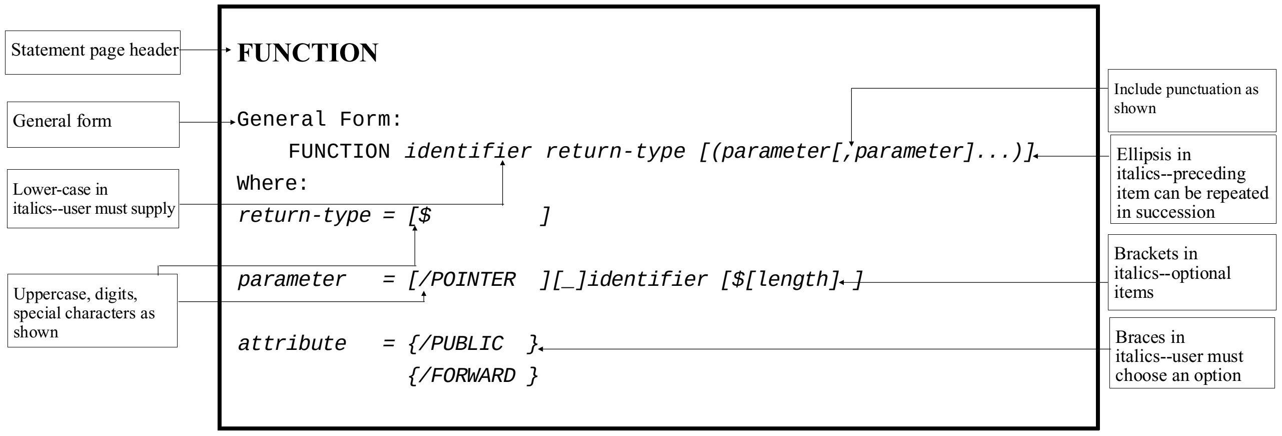
  For example:

  ```
  DEFFN'integer[(variable[,variable]...)]
  ```

  Here, any number of "variable" may be specified, but the format ",variable" must be used for the second and subsequent "variables".

- All other punctuation such as commas or parentheses must be included where shown.

Here is an illustration of statement conventions.



FUNCTION

```
General Form:
    FUNCTION identifier return-type [(parameter[,parameter]...)]
Where:
return-type = [$        ]

parameter   = [/POINTER  ][_]identifier [$[length] ]

attribute   = {/PUBLIC  }
              {/FORWARD }
```

Statement page header
General form
Lower-case in italics--user must supply
Uppercase, digits, special characters as shown
Include punctuation as shown
Ellipsis in italics--preceding item can be repeated in succession
Brackets in italics--optional items
Braces in italics--user must choose an option

Illustration of Statement Conventions

1.2.4 Terminology

The following terms are commonly used and are specifically related to the statements described in Chapter 2 of this manual. Those terms not found in this section are defined by the "Where:" clause for the individual statement.

address-var:
An alpha-variable in which the first three bytes contain an ASCII representation of a hex-digit (0-9; A-F) which corresponds to a NPL device address must be specified.

alpha-array:
An alpha array variable must be specified.

alpha-expression:
Any valid alpha expression may be used. This includes the use of literals as well as alpha functions. Refer to LET for further details on alpha expressions.

alpha-receiver:
A series of alpha variables delimited by commas. At least one alpha variable must be specified. When multiple variables are specified, all are set to the value of the expression on the right side of the equivalence.

alpha-variable:
An alpha variable, either scalar or array, alpha array element, STR function, or string field-expression must be specified.

array-variable:
A numeric or alpha array variable must be specified.

device-address:
A valid NPL device address in the format /xxx where x is a hex-digit must be specified. Refer to Chapter 7 of the NPL Programmer's Guide for details on NPL device addresses.

diskimage:
Refers to a NPL diskimage file or a "raw" disk device. Refer to Chapter 7 of the NPL Programmer's Guide for details on diskimage files and "raw" devices.

file-number:
A valid NPL file number in the format #X where 0 <= X <= 15 must be specified. File numbers above 15 require the appearance of a SELECT #n statement in the program. Refer to Chapter 7 of the NPL Programmer's Guide for details on file numbers.

hex-digit:
A hexadecimal digit (0-9, A-F) must be specified.

identifier:
Any legal (long) identifier. Up to 255 alphanumeric characters (A-Z, a-z, 0-9 or _), starting with a letter (digits, _ not permitted as first character).

line-number:
A valid NPL program line number must be specified. Valid values are in the range 0 to 32117.

literal-string:
An alpha literal string must be specified. This may be either a string (enclosed in quotes) or a HEX literal.

logical-expression:
An conditional expression which evaluates to either true or false, used when making decisions. Refer to the IF/THEN statement for further details on logical expressions.

numeric-array:
A numeric array variable must be specified. A specific element of a numeric-array may not be specified.

numeric-constant:
A numeric constant must conform to the format:

```
[+]d...[.[d]...][E[+]d[d]]
[+]             [-]
```

where:

d is a digit (0-9). Up to 13 digits may be specified.
+ or - are signs and may be either leading or trailing.
E is the exponent symbol (may be preceded by an optional sign and followed by one or two digits).
. is a decimal point.

numeric-expression:
A valid numeric expression must be specified. This may include numeric constants, functions, and variables. Refer to LET for further details on numeric expressions.

numeric field-expression:
An alpha-variable followed by a "." and a numeric field identifier. The numeric field identifier may be either a scalar field or an array element. A numeric field-expression refers to a single numeric field of a record, using the named alpha-variable as a buffer for the record.

numeric-receiver:
A numeric scalar or numeric array element used to receive the results of an operation.

numeric-scalar:
A valid numeric scalar variable must be specified.

numeric-variable:
A valid numeric variable, either scalar or array, must be specified.

simple-statement:
A simple statement is a statement that can stand alone as a complete program. Some statements, in particular those used to implement structured programming constructs, are incomplete in that they require a corresponding statement to terminate them (e.g., WHILE requires WEND).

statement-label:
An identifier which is appears elsewhere in the program as a statement label (=identifier).

string field-expression:
An alpha-variable followed by a "." and a string field identifier. The string field identifier may be either a scalar field or an array element. A string field-expression refers to a single string field of a record, using the named alpha-variable as a buffer for the record.

1.2.5 Variable Names

The following variable names are used with the NPL statements defined in this manual.

NOTE: **For historical reasons, short identifiers (a letter followed by 0-9 or a number in the range 10-62 with no leading zeroes) used for numeric scalar, alpha scalar, numeric array, and string array variables are always displayed in upper case.**

Numeric Scalar

Numeric scalar variables names consist of an identifier. Some valid numeric scalar variable names are:

A	D2	X4	Apple
TimeOfDay	DAY_OF_WEEK		
APPLE	window	Stalag_17	uP_aNd_DoWn
Cost_Of_Building_When_Renovations_Are_Complete_Not_Including_Tax			

Alpha Scalar

Alpha scalar variable names consist of an identifier followed by a dollar sign ($). Some valid alpha scalar variable names are:

A$	D2$	X4$	Apple$
TimeOfDay$	DAY_OF_WEEK$		
APPLE$	window$	Stalag_17$	uP_aNd_DoWn$
Name_Of_That_Singer_Who_Looks_Like_Julio_Iglesias_But_Isnt_Actually_Him$			

Constant Scalars

Constant scalar names consist of an underline (_) followed immediately by a scalar variable name. Some valid constant scalar names are:

_A	_D2	_X4	_Apple$
_TimeOfDay	_DAY_OF_WEEK		
_APPLE	_window	_Stalag_17	_uP_aNd_DoWn
_Cost_Of_Building_When_Renovations_Are_Complete_Not_Including_Tax			

NOTE: The variable _x is not the same as x. _x is defined as a constant variable, where x is a numeric scalar and NPL treats them as such.

Numeric Array

Numeric array variable names consist of an identifier followed by a set of parentheses. Some valid numeric array variable names are:

A()	D2()	X4()	Apple()
TimeOfDay()	DAY_OF_WEEK()		
APPLE()	window()	Stalag_17()	uP_aNd_DoWn()
LongNumericArrayExample()			

NOTE: Array names are distinct from scalar names. For example, A() and A are distinct variable names.

Exceptions:
Several MAT statements allow reference to an entire array by designation of the root name with no parentheses. For example:

```
MAT A = C + B
```

refers to numeric arrays A(), B(), and C().

Several listing or debugging type statements allow reference to numeric arrays by specification of the root variable name followed by a single open parenthesis. For example:

```
LIST V A(
```

will list all references to numeric array A().

Numeric Array Elements

A numeric array element consist of an identifier followed by one or more subscripts in parentheses. If a one dimensional array is used, only a single subscript may be present. If a two dimensional array is used, two subscripts must be present separated by a comma. Some valid numeric array elements are:

A(1)	D2(2,3)	X4(5)	Apple(2,4)
TimeOfDay(2,2)	DAY_OF_WEEK(7)		
APPLE(6)	window(3)	Stalag_17(6,2)	
uP_aNd_DoWn(4)	LongNumericArrayExample(6,6)		

Alpha Array

Alpha array variable names consist of an identifier followed by a dollar sign, followed by a set of parenthesis "()". Some valid alpha array variable names are:

A$()	D2$()	X4$()	Apple$()
TimeOfDay$()	DAY_OF_WEEK$()		
APPLE$()	window$()	Stalag_17$()	uP_aNd_DoWn$()
Names_Of_All_The_Crew_Who_Are_Of_Scottish_Ancestry$()			

NOTE: Array names are distinct from scalar names. For example, A$() and A$ are distinct variable names.

Exceptions:
Several MAT statements allow reference to an entire array by designation of the root name with no parentheses. For example:

```
MAT PRINT A$,B$
```

refers to alpha array A$(), and B$().

Several listing or debugging type statements allow reference to alpha arrays by specification of the root variable name followed by a single open parenthesis. For example:

```
LIST V A$(
```

will list all references to alpha array A$().

Alpha Array Elements

An alpha array element consists of an identifier, followed by a dollar sign ($) followed one or more subscripts (numeric expressions) in parentheses. If a one-dimensional array is used, only a single subscript may be present. If a two-dimensional array is used, two subscripts must be present, separated by a comma. Some valid alpha array elements are:

```
A$(1)                           D2$(2,3)        X4$(5)
Apple$(X+Y)                     TimeOfDay$(Local)
DAY_OF_WEEK$(Index)             APPLE$(I0)      window$(pane)
 Stalag_17$(William,Holden)     uP_aNd_DoWn$(sLiNky(Toy)*SQR(5))
 Spread_Sheet_With_All_My_Personal_Information$(Row,Column)
```

Constant Array

Constant array variable names consist of an underline (_) followed by an identifier and set of parentheses. Some valid constant array names are:

```
_A()              _D2()            _X4()           _Apple$()
_TimeOfDay()      _DAY_OF_WEEK()
_APPLE()          _window()        _Stalag_17()    _uP_aNd_DoWn()_
 Coordinates_Where_The_ENTERPRISE_Completed_Its_5_Year_Mission()
```

Constant Array Elements

A constant array element consists of an underline (_), followed by an identifier and one or more subscripts (numeric expressions) in parentheses. If a one-dimensional array is used, only a single subscript may be present. If a two-dimensional array is used, two subscripts must be present, separated by a comma. Some valid constant array elements are:

```
_A(1)                               _D2(2,3)          _X4(5)
_Apple$(X+Y)                        _TimeOfDay(Local)
_DAY_OF_WEEK(Index)                 _APPLE$(I0)      _window(pane)
 _Stalag_17\(William,Holden)        _uP_aNd_DoWn(sLiNky(Toy)*SQR(5))
 _Spread_Sheet_With_All_My_Personal_Information(Row,Column)
```

Numeric field-expression

A string field identifier consists of a "." followed by either a numeric or numeric array element. A numeric field-expression consists of an alpha-variable followed by a numeric field identifier. Some valid numeric field-expressions are:

```
Employee_Record$.Number_Of_Children
Employee_Record$.YTD_Misc_Deduction(1)
Employee_Record$.Department_Number
Deduction_Table$(X).Limit
 STR(Buffer$,P).Header_Extension$.Header_Extension_Length
```

String field-expression

A string field identifier consists of a "." followed by either an alpha scalar or alpha array element. A string field-expression consists of an alpha-variable followed by a string field identifier. Some valid string field-expressions are:

```
Employee_Record$.Employee_Name$
Employee_Record$.Address1$
Employee_Record$.Child_name$(1)
Deduction_Table$(X).Description$
STR(Buffer$,P).Header_Extension$.Header_Extension_ID$
```

1.3 Organization of the Statements Guide

This Statements Guide is divided into the following chapters:

- Chapter 1 explains how information has been presented in this guide.
- Chapter 2 contains detailed information and instructions on using NPL statements.
- Chapter 3 describes the operation of the library functions available in the NPL.

In addition, to the sections listed above, this Statements Guide contains two appendices . Appendix A lists reserved words and Appendix B describes NPL/Basic-2 statement compatibility.

CHAPTER 2

LANGUAGE STATEMENTS

2.1 Overview

This chapter contains descriptions, examples and other detailed information pertaining to the NPL statements. Individual discussions of NPL statements, listed in alphabetical order, begin on the following page.

ABS Function

General Form:

`ABS`*`(numeric-expression)`*

Discussion:

The ABS function computes the absolute value of a numeric-expression. This is valid wherever a numeric function is legal.

Examples:

```
:0010 PRINT "RESULT=";ABS(-4.75)
:RUN
RESULT= 4.75

:0010 X=-10
:0020 PRINT "RESULT=";5*ABS(X)
:RUN
RESULT= 50
```

Compatibility Issues:

References:

ADD[C] Alpha-operator

General Form:

```
     alpha-receiver = [...] ADD[C] alpha-operand [...]
```

Where:

```
     alpha-operand = {literal-string  }
                     {alpha-variable  }
                     {ALL function    }
                     {BIN function    }
                     {system-variable }
```

Discussion:

The ADD alpha-operator is used to add the binary value of an alpha-operand to the binary value of an alpha-variable. ADD may only be used in an alpha-expression in an alpha-assignment statement.

Each byte of alpha-operand is ADDed to each corresponding byte of the receiving alpha-variable. The ADD operation is performed from right to left, starting with the right-most byte. If "C" immediately follows the ADD alpha-operator, then carry propagation is effected between bytes to yield full multi-byte binary number addition.

If the value of alpha-operand and the receiving alpha-variable are of different lengths, then the ADD algorithm implicitly extends the shorter value with leading zeroes prior to ADDing. If the ADD resultant is larger than the receiving alpha-variable, then the extraneous high-order bytes of the resultant are truncated before assignment.

NOTE: Contrary to conventional alpha-variable operations, the ADD alpha-operator operates on all bytes of an alpha-variable (either as a receiver or alpha-operand), including trailing spaces.

The ADD[C] alpha-operator is often used in conjunction with SUB[C], BIN and VAL.

ADD[C] (cont.)

Examples:

```
0010  A$=ADD B$
0010  A$=ADD ALL(01)
0010  A$=B$ ADD C$
0010  STR(A$,3,2)=ADD X$
0010  X$=ADDC HEX(00FF)
0010  Myrec$.Field$=ADD ALL(0F)
:0010 DIM A$2
:0020 A$=HEX(0121)
:0030 A$=ADD HEX(00FF)
:0040 PRINT "A$=";HEXOF(A$)
:RUN
A$=0120
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

BIN
SUB[C]
VAL

+=numeric expression Add to Variable Statement

General Form:

```
     numeric-var += numeric-expression
```

Where:

```
     numeric-var          = a valid numeric variable (i.e., sca-
                            lar or array element)

     numeric-expression = a valid numeric expression
```

Discussion:

The add to variable statement avoids the repetition of long variable names in common increment uses (it is not intended to be faster than the common add).

This is not a numeric operator. It can only appear as a statement by itself.

NOTE: **Only one variable is permitted on the left-hand side of the +=, but it may be either a scalar or an array element.**

Examples:

```
0010 I+=1
0010 I+=Array(X,Y)
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

LET Numeric Assignment
-=

$ALERT

General Form:

```
$ALERT partition-number
```

NOTE: This statement is supported for Wang compatibility reasons only and its use in new development is not recommended.

Discussion:

The syntax of this statement is supported only for compatibility with Wang 2200 Basic-2. No operation is performed when this statement is encountered at execution time.

The compiler generates a warning when this statement is encountered.

Examples:

Compatibility Issues:

This instruction, when executed on a Wang 2200 MVP, signals the specified partition to execute the subroutine specified by its programmable interrupt table for the ALERT condition, provided such a subroutine has been defined, and interrupts are enabled in the target partition. The subroutine is performed at the next breakpoint in the partition's execution.

Interrupts are not supported by NPL.

References:

ALL Alpha-operand

General Form:

```
     ALL ({literal-string  })
          {alpha-variable  }
          {two-hexdigits   }
```

Discussion:

The ALL function creates a temporary character string of unlimited length with each character of the string equal to the character specified in the function. The ALL function may be used as an alpha-operand to any alpha-operator and it is legal only in an alpha-expression in an alpha-assignment statement and in no other statement in the language.

The character to be used by the ALL function can be specified as a literal-string, an alpha-variable or as a pair of hex digits (0-9 or A-F). If an alpha-literal or alpha-variable is specified, only the first character is used by the ALL function.

The ALL function is useful for initializing alpha-variables (scalars and arrays).

Examples:

```
0010 A$=ALL(B$)
0010 A$=ALL(STR(B$,4,1))
0010 A$=ALL(" ")
0010 A$=C$ AND ALL(F0)

:0010 DIM A$5
:0020 A$=ALL(40)
:0030 PRINT A$
:RUN
@@@@@

:0010 DIM A$16
:0020 A$="AND SO ON" & ALL(".")
:0030 PRINT A$
:RUN
AND SO ON.......

0010 MyRec$.field1$ = ALL(20)
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

AND Alpha-operator

General Form:

```
     alpha-receiver =  [...] AND alpha-operand [...]
```

Where:

```
     alpha-operand =  {literal-string  }
                      {alpha-variable  }
                      {ALL function    }
                      {BIN function    }
                      {system-variable }
```

Discussion:

The AND logical alpha-operator performs a logical AND operation on the alpha-operand and the contents of the alpha-receiver, the result of which is then assigned to the alpha-receiver. The AND alpha-operator is legal only in an alpha-expression in an alpha-assignment statement.

The AND operation is performed on a byte-by-byte basis, moving from left to right in each field, for a number of bytes equal to the shorter of:

- The defined length of the alpha-receiver.
- The defined length of the alpha-operand (if the alpha-operand is an alpha-variable or system-variable, trailing spaces are included in the operation).

If the defined length of the alpha-operand is shorter than the defined length of the alpha-receiver, then the remaining bytes of the alpha-receiver remain unchanged (i.e., padding with spaces is not performed).

NOTE: In regard to the "AND" syntactic unit, this may also appear in conditional-expressions (e.g., IF A=1 AND B=2 THEN ...). However, the similarity is syntactical only and its use in a conditional-expression has a completely different meaning.

AND Alpha-operator (cont.)

Examples:

```
0010 MyRec$.Field2$=AND B$
0010 STR(A$,4,5)=AND B$
0010 A$=C$ AND "0"

:0010 DIM A$5
:0020 A$=ALL(FF)
:0030 A$=AND HEX(7F)
:0040 PRINT HEXOF(A$)
:RUN
7FFFFFFFFF
```

In statement 30 of the above example, the defined length of A$ is 5, the length of the operand (HEX(7F)) is one; therefore, only the first byte of A$ is ANDed and all remaining bytes are unchanged.

```
:0010 DIM A$5
:0020 A$=ALL(FF)
:0030 A$=AND ALL(7F)
:0040 PRINT HEXOF(A$)
:RUN
7F7F7F7F7F
```

In statement 30 of this example, the defined length of A$ is 5, but the length of the operand (ALL(7F)) is unlimited; therefore, all bytes of A$ are ANDed.

Compatibility Issues:

This statement is supported only with Release IV or greater.

References

BOOL

ARC COS Function

General Form:

`ARC COS`*`(numeric-expression)`*

Discussion:

The ARC COS function computes the value of the arccosine of a numeric-expression. This is valid wherever a numeric function is legal.

The calculation is performed in Degrees, Radians, or Gradians, depending on last execution of SELECT [D,R,G] statement.

Examples:

```
:0010 A=.25
:0020 B=ARC COS(A)*10
:0030 PRINT "RESULT="; B
:RUN
RESULT= 13.181160716528
```

Compatibility Issues:

Due to the use of a different algorithm, results of these functions may differ from functions evaluated on a Wang 2200. In general, however, the functions are accurate to 13 significant digits.

References:

SELECT [D,R,G]

ARC SIN Function

General Form:

`ARC SIN`*`(numeric-expression)`*

Discussion:

The ARC SIN function computes the value of the arcsine of a numeric-expression. This is valid wherever a numeric function is legal.

The calculation is performed in Degrees, Radians, or Gradians, depending on last execution of SELECT [D,R,G] statement.

Examples:

```
:0010 A=.25
:0020 B=ARC SIN(A*.25)*2
:0030 PRINT "RESULT="; B
:RUN
RESULT= .12508152359299
```

Compatibility Issues:

Due to the use of a different algorithm, results of these functions may differ from functions evaluated on a Wang 2200. In general, however, the functions are accurate to 13 significant digits.

References:

SELECT [D,R,G]

ATN Function - ARC TANGENT

General Form:

```
ATN(numeric-expression)
```

Discussion:

The ATN function computes the value of the arctangent of a numeric-expression. This is valid wherever a numeric function is legal.

The calculation is performed in Degrees, Radians, or Gradians, depending on last execution of SELECT [D,R,G] statement.

Examples:

```
:0010 A=.125
:0020 B=ATN(A*.25)+1.5
:0030 PRINT "RESULT="; B
:RUN
RESULT= 1.5312398334303
```

Compatibility Issues:

Due to the use of a different algorithm, results of these functions may differ from functions evaluated on a Wang 2200. In general, however, the functions are accurate to 13 significant digits.

References:

SELECT [D,R,G]

BIN Function/Alpha-operand

General Form:

```
BIN(numeric-expression[,range-expression])
```

Where:

```
range-expression = a numeric-expression with a result be-
                   tween -6 and +5.
```

Discussion:

The BIN function is used to convert the integer result of a numeric-expression into character string format, binary representation. The resulting character string may then be used as an alpha-operand to any alpha-operator in an alpha-expression. The BIN function may only be used in an alpha-expression in an alpha-assignment statement and cannot be used in any other statement in NPL. The BIN function is most useful for conversion of numbers stored in internal numeric format to binary for special manipulation or use.

The range-expression of the BIN function is used to specify both the length and content of the resultant character string. The range-expression must evaluate to a number from -6 to +5, otherwise an error results. If the range-expression is omitted, a value of 1 is assumed.

The absolute value of the range-expression indicates the length of the resultant character string to be generated by BIN. A length from 0 bytes up to 6 bytes is acceptable.

The sign of the range-expression value indicates the type of binary number to be generated in the character string. If the sign is positive (+), an unsigned binary integer is generated. If the sign is negative (-), a signed, two's complement, binary integer is generated.

An error is generated if the numeric-expression cannot be fully represented within a character string of the selected length. The following table summarizes the range of numbers which can be converted for each possible value of the range-expression.

BIN Function/Alpha-operand (cont.)

Range Expression	Resultant Length (bytes)	Type	Range allowed for numeric-expression	
-6	6	signed	-140737488355328	140737488355327
-5	5	signed	-549755813888	549755813887
-4	4	signed	-2147483648	2147483647
-3	3	signed	-8388608	8388607
-2	2	signed	-32768	32767
-1	1	signed	-128	127
0	0	unsigned	0	0
1	1	unsigned	0	255
2	2	unsigned	0	65535
3	3	unsigned	0	16777215
4	4	unsigned	0	4294967295
5	5	unsigned	0	1099511627775

Examples:

```
0010  A$=BIN(X/2,3)
0010  A$=BIN(X)
0010  A$=BIN(X,-(1+Y))

:0010 X$=BIN(65)  : REM BINARY VALUE OF DECIMAL 65 IS ASCII "A"
:0020 PRINT X$;" ";HEXOF(X$)
:RUN
A 41202020202020202020202020202020

:0010 B$=BIN(18505,2)
:0020 PRINT B$;" ";HEXOF(B$)
:RUN
HI 48492020202020202020202020202020
```

Compatibility Issues:

On the Wang 2200, the BIN function generates up to a maximum two byte unsigned character string only. Furthermore, on the Wang 2200, the second BIN operand, if specified, must be a ",2", numeric-expressions are not allowed in NPL.

References:

VAL

BOOL Alpha-operator

General Form:

```
    alpha-receiver = [...] BOOLh alpha-operand [...]
```

Where:

```
    h              = hexadecimal digit (0-9 or A-F)

    alpha-operand = {literal-string  }
                    {alpha-variable  }
                    {ALL function    }
                    {BIN function    }
                    {system-variable }
```

Discussion:

The BOOL logical alpha-operator performs the specified logical operation on the alpha-operand and the contents of the alpha-receiver, the result of which is then assigned to the alpha-receiver. The BOOL alpha-operator is legal only in an alpha-expression in an alpha-assignment statement.

The BOOL logical operation is performed on a byte-by-byte basis, moving from left to right in each field, for a number of bytes equal to the shorter of:

- The defined length of the alpha-receiver.
- The defined length of the alpha-operand (if the alpha-operand is an alpha-variable or system-variable, trailing spaces are included in the operation).

If the defined length of the alpha-operand is shorter than the defined length of the alpha-receiver, then the remaining bytes of the alpha-receiver remain unchanged (i.e., padding with spaces is **not** performed).

The character immediately following BOOL represents the logical operation to be performed. For example, BOOL**7** would specify that a "not-AND" operation be performed. The table on the following page lists the available hex digits with their corresponding logical functions.

BOOL Alpha-operator (cont.)

BOOLh Logical Functions		
Hex Digit	**Binary Representation**	**Logical Function**
0	0000	Null
1	0001	Not-OR
2	0010	Operand does not imply receiver
3	0011	Complement of receiver
4	0100	Receiver does not imply operand
5	0101	Complement of operand
6	0110	Exclusive OR
7	0111	Not-AND
8	1000	AND
9	1001	Equivalence
A	1010	Receiver = operand
B	1011	Receiver implies operand
C	1100	Operand = receiver
D	1101	Operand implies receiver
E	1110	OR
F	1111	Identity

When working with complicated boolean functions, it is not necessary to memorize the order of the 16 BOOL functions. The appropriate function to use can be easily determined by filling in the following truth table:

Bit in receiver is a	1	1	0	0
Bit in operand is a	1	0	1	0
Required result bit =	(w)	(x)	(y)	(z)

The resulting bit pattern wxyz specifies the correct Hex-digit to use as a BOOL function.

BOOL Alpha-operator (cont.)

For example, to "zero each bit in the receiver where the corresponding bit is a 1 in the operand", the truth table would be filled in as follows:

Bit in receiver is a	1	1	0	0
Bit in operand is a	1	0	1	0
Required result bit =	0	1	0	0

Examples:

The resulting four bits 0100 specify that BOOL4 is the appropriate function to use. Using this technique, it is clear that AND is equivalent to BOOL8 (result 1000) and OR is equivalent to BOOLE (result 1110).

```
0010  A$=BOOL7 C$
0010  X$=BOOL1 Y$
0010  L$=BOOL9 ALL(7F)
0010  STR(X$,3,2)=BOOL1 B$

:0010 DIM X$2
:0020 X$=HEX(1100) BOOL9 HEX(1010) AND HEX(1111)
:0030 PRINT HEXOF(X$)
:RUN
1001
```

Compatibility Issues:

References:

AND

$BOXTABLE

General Form:

Form 1

```
alpha-receiver=$BOXTABLE
```

Form 2

```
$BOXTABLE=alpha-expression
```

Discussion:

Form 1 of the $BOXTABLE statement sets the value of the $BOXTABLE system variable.

Form 2 allows examination of the current status of the $BOXTABLE system variable.

The $BOXTABLE system variable is used to enable, and select the character set to be employed for the output of the PRINT BOX statement. Two forms of boxes are supported: "True" boxes and "Character" boxes.

True Box Graphics

"True" box graphics require a screen which has the capability of printing graphics and text on the same screen (refer to the appropriate operating system-specific supplement for this information). The implementation of "character" box graphics provides a method of approximating box graphics on machines which do not have this capability.

Character Box Graphics

"Character" box graphics are built from the standard character set. That is, 16 characters must be selected out of the standard character set to be used for the 16 possible "box" characters (appropriate default values are provided; refer to the table which follows). The use of character box graphics, therefore, has some significant limitations. The primary limitation is that the horizontal lines of a box cannot be printed between rows (as they are with "true" boxes), but rather must occupy a row themselves. The horizontal lines are printed a half-line down from where they would print using "true" boxes. This means that, in effect, one line above and below the text to be boxed must be left blank.

$BOXTABLE (cont.)

Byte 4 of the $MACHINE system variable can be used by NPL application programs to determine whether or not "true" box graphics are available. For further details, refer to $MACHINE.

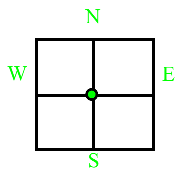
The $BOXTABLE system variable consists of 17 bytes. Byte 1 is a switch which the RunTime program tests to determine whether or not to print "character" boxes. A value of HEX(00) (the default) indicates that "character" boxes are disabled. A value of HEX(01) indicates that "character" boxes are enabled.

NOTE: If character boxes are enabled, they are printed even if "true" box graphics are available (the "true" boxes are not printed in this case).

Construction of "Character" Boxes

Each individual character of a "character" box should consist of some combination of four basic parts. These four parts are:

- Vertical line from center of character to north edge (N)
- Vertical line from center of character to south edge (S)
- Horizontal line from center of character to east edge (E)
- Horizontal line from center of character to west edge (W)



Combinations of these four basic parts (illustrated above) are required to display the complete box character required. Up to 16 combinations are possible. When the RunTime program prints a "character" box, it calculates the correct combination to use for each character position of the box. It then determines the proper character to use by means of a look-up table. Bytes 2 to 17 of the $BOXTABLE system variable contain the actual characters to be used for each of the 16 possible combinations used to construct a box (refer to the table below for default values).

$BOXTABLE (cont.)

These 16 possible values relate to byte positions in the $BOXTABLE as follows:

Byte	**N**	**S**	**E**	**W**
2	.	.	.	.
3	.	.	.	X
4	.	.	X	.
5	.	.	X	X
6	.	X	.	.
7	.	X	.	X
8	.	X	X	.
9	.	X	X	X
10	X	.	.	.
11	X	.	.	X
12	X	.	X	.
13	X	.	X	X
14	X	X	.	.
15	X	X	.	X
16	X	X	X	.
17	X	X	X	X

For example, the character in byte 5 would be used for a horizontal line from the west edge to the east edge of the character. The character in byte 14 would be use for a vertical line. The character in byte 8 would be used for the upper left hand corner.

Text and character boxes may not occupy the same character position on the screen. Therefore, printing text (except spaces) on top of a box erases that portion of the box. Printing boxes on top of text does not overwrite that portion of text. Consequently, character boxes may be printed before or after text with the same result.

$BOX TABLE (cont.)

Examples:

```
0010 DIM X$17,M$4       :; variable for table must be 17
                           bytes long
0020 M$=$MACHINE        :; get system information
0030 X$=$BOXTABLE       :; load current values
0035 ;if not 'TRUE' graphics then modify $BOXTABLE
0040 IF STR(M$,4,1) <> "G"
0045   ; enable character boxes with default box character set
0050   $BOXTABLE=BIN(1) & STR(X$,2)
0060 END IF
0070 RETURN
```

HINT: Restore the value of $BOXTABLE to the original value by programs which modify it so that use of character boxes is possible on a modularized basis. In general, some programs can easily be modified to work with character boxes while, in other cases, it is preferable to leave the boxes off entirely.

Compatibility Issues:

This statement is supported only with Release 1.03 or greater.

This statement is not valid in Wang 2200 Basic-2.

The default values for the characters to use for character boxes (bytes 2-17) vary depending upon the hardware version of NPL. Refer to the appropriate NPL Supplement for details.

References:

$MACHINE
PRINT BOX
Box Graphics - Section 7.3.19 of the Programmer's Guide

BREAK

General Form:

```
BREAK
```

Discussion:

The BREAK statement allows exiting from the body of a structured loop, which may be either WHILE...WEND, REPEAT...UNTIL or FOR/BEGIN...NEXT type. When it occurs inside nested loops, only the innermost loop is exited.

When executed, control is transferred to the statement following the WEND statement of the current WHILE...WEND loop, to the statement following the UNTIL statement of the current REPEAT...UNTIL loop or to the statement following the NEXT statement of an enclosing FOR/BEGIN...NEXT loop. Stack information is cleared for a FOR/BEGIN...NEXT loop.

Examples:

```
0010;
  :X=1
  :REPEAT
  :    X=X+X
  :     IF X>1000 THEN BREAK
  :    PRINT X;
  :UNTIL FALSE
  :;
  :X = 1
  :WHILE TRUE
  :  X = X + X
  :  IF X > 1000 THEN BREAK
  :  PRINT X;
  :WEND
  :;
  :FOR T = 1 TO 20 BEGIN
  :  X = 'Approx(X)
  :  IF 'Funct(X) < Epsilon THEN BREAK
  :NEXT T
```

Compatibility Issues:

This statement is only supported with Release IV or greater.

BREAK (cont.)

References:

FOR/BEGIN ... NEXT
REPEAT/UNTIL
WHILE/WEND
Section 4.11 of the Programmers Guide.

$BREAK

General Form:

```
$BREAK [expression]
```

Discussion:

$BREAK provides a mechanism in a multi-user environment for relinquishing CPU timeslices from the program in progress. The number of timeslices released is the integer portion of the expression. The numeric-expression can be a value from 0 to 255. The default value, if no expression is specified, is one. A value of zero indicates that no break occurs.

The $BREAK statement is primarily used to put a user partition to sleep until a given occurrence.

The actual effect of $BREAK [expression] is hardware and operating system-dependent. On most single-user systems, no time is released. On multi-user systems, the amount of time released per unit may vary. Refer to the appropriate NPL Supplement for details.

NOTE: **The $BREAK statement is extremely operating system-dependent. Refer to the NPL Supplements for details on the operation of $BREAK on different platforms.**

Examples:

```
0010 $BREAK 5
0010 $BREAK X+Z
```

Compatibility Issues:

On a Wang 2200 MVP, the "!" parameter causes the partition to relinquish all timeslices until the RESET key is pressed, or a programmable interrupt occurs. $BREAK! is decompiled under NPL as $END, causing immediate exit from the RunTime program. In addition, the amount of time released on a Wang 2200 is dependent on other activity on the system. Under NPL, a fixed amount of time per unit is released.

In a network environment, the syntax is supported but no operation is performed.

References:

CASE Default

General Form:

```
CASE
```

Discussion:

This statement declares a default condition of a numeric, string or logical SWITCH structure. Refer to SWITCH (numeric, string or logical) for an explanation of how this statement may be used within a SWITCH structure to define the default action to take when no specific CASE applies.

Examples:

```
0010 ;
:SWITCH Widget_Type
:   CASE 0
:      PRINT "Gizmos"
:   CASE 1
:      PRINT "Thingammies"
:   CASE
:      ; this is the default
:      PRINT "Whatchamacallits"
:END SWITCH
```

NOTE: Branching to a CASE statement is not the same as reexecuting the CASE. Executing a CASE statement terminates the previous case, and so exits to the END SWITCH statement. This is an easy mistake to make when converting an ON X statement to a SWITCH that still contains GOTOs used for loops.

For example, this is old code:

```
0000 ON X GOTO 100,200,300      :REM lookup type (assume X=1,2,3 only)
0100 INPUT "Name",A$:IF A$" " THEN 900
    :  GOSUB 'Doname
    :  GOTO 100
0200 INPUT "Address",A$:IF A$=" " THEN 900
    :  GOSUB 'DoAddress
    :  GOTO 200
0300 INPUT "Zip",A$:IF A$=" " THEN 900
    :  GOSUB 'DoZip
    :  GOTO 300
0900 REM end case
```

CASE Default (cont.)

This is not the same as:

```
0000 SWITCH X                 :REM lookup type (assume X=1,2,3 only)
0100 CASE 1
    :  INPUT "Name",A$:IF A$=" " THEN 900
    :  GOSUB 'Doname
    :  GOTO 100
    :CASE 2
0200   INPUT "Address",A$:IF A$=" " THEN 900
    :  GOSUB'DoAddress
    :  GOTO 200
    :CASE 3
0300   INPUT "Zip",A$:IF A$=" " THEN 900
    :  GOSUB 'DoZip
    :  GOTO 300
0900 END SWITCH
```

Correct is:

```
0000 SWITCH X          :REM lookup type (assume X=1,2,3 only)
    :CASE 1
0100    INPUT "Name",A$:IF A$=" "THEN 900
    :   GOSUB 'Doname:GOTO 100
    :CASE 2
0200    INPUT "Address",A$:IF A$=" "THEN 900
    :   GOSUB 'DoAddress
    :   GOTO 200
    :CASE 3
0300    INPUT "Zip",A$
    :   IF A$=" "THEN 900
    :   GOSUB 'DoZip
    :   GOTO 300
0900 END SWITCH
```

Compatibility Issues:

This statement is only supported with Release IV or greater.

References:

END SWITCH
SWITCH Default
Logical Constructs, Section 4.11 of the NPL Programmer's Guide

CASE Logical

General Form:

CASE *logical-expression[,logical-expression]*

Where:

logical-expression = {cond [logical-operator cond]...}

Discussion:

This statement may only occur within a logical SWITCH structure. Refer to SWITCH Logical for an explanation of how this statement may be used within a logical SWITCH structure to define the action to take for specific logical case values.

Examples:

The following is an example of valid syntax:

```
0010 CASE Machine$ = "I", Machine$ = "W"
0010 CASE Machine$ = "I" OR Machine$  = "W"
0010 CASE Index < CacheIndex
```

The following is an example for logical default CASE:

```
0010 number = RND(1023)
   : SWITCH
   :  CASE number <= 0.25
   :   PRINT "0.0 <= ";number;" <= 0.25"
   :  CASE number <= 0.5
   :   PRINT "0.25 < ";number;" <= 0.5"
   :  CASE number <= 0.75
   :   PRINT "0.5 < ";number;" <= 0.75"
   :  CASE
   :    ; default, always the last CASE statement
   :   PRINT "0.75 < ";number;" <= 1.0"
   : END SWITCH
```

CASE Logical (cont.)

```
0010 SWITCH
    :    CASE char$="a" OR char$="b",char$="c"
    :        PRINT "a, b or c"
    :    CASE char$> "c" AND char$<"m"
    :        PRINT "d through l"
    :    CASE
    :        ; this is the default
    :        PRINT "m through z"
    :  END SWITCH
0020  SWITCH
    :   CASE Error_Code=48 OR Error_Code=37
    :      Errtype = _RECOVERABLE    :; changed our minds about these
    :   CASE Error_Code<60
    :      Errtype = _UNRECOVERABLE  :; syntax and programming errors
    :   CASE Error_Code<100
           Errtype = _RECOVERABLE    :; Range errors, I/O errors
    :   CASE Error_Code<200
    :      Errtype =_RESERVED        :;Future use
    :   CASE Error_Code<300
    :      Errtype =_UNRECOVERABLE   : ;Extended errors
    :   CASE Error_Code<500
    :      Errtype=_RECOVERABLE      : ;Extended errors
    :   CASE Error_Code<600
    :      Errtype=_UNRECOVERABLE    : ;External errors
    :   CASE Error_Code<800
    :      Errtype=_RECOVERABLE      : ;External errors
    :   CASE
    :      Errtype=_RESERVED         : ;Future use
    : END SWITCH
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

Logical Constructs - Section 4.11 of Programmer's Guide
SWITCH Logical

CASE Numeric

General Form:

```
CASE numeric-expression[,numeric-expression]...
```

Discussion:

This statement may only occur within a numeric SWITCH structure. Refer to SWITCH Numeric for an explanation of how this statement may be used within a numeric SWITCH structure to define the action to take for specific case values.

Examples:

An example of valid syntax is shown below.

```
0010 CASE 0
0010 CASE 0,1,2,12,4
0010 CASE _PACK_IBMASCII_FORMAT
0010 CASE X(T),X(T2),X(T3)
0010 CASE 'LeftButton
```

An example of practical usage of the statement is shown below.

```
0010 ;
    : SWITCH number
    :    CASE 1, 2, 4, 8, 16
    :        PRINT "powers of 2"
    :    CASE _PACK_IBMASCII_FORMAT
    :        PRINT "a format specification"
    :    CASE array(T)
    :        PRINT "an array element"
    :    CASE 'real_random_num
    :        PRINT "not very random"
    :    CASE
    :        PRINT "try another number"
    : END SWITCH
```

CASE Numeric (cont.)

The following is an example of numeric default CASE:

```
0010 SWITCH day_of_week
    : CASE 1
    :    PRINT "Monday"
    : CASE 2
    :    PRINT "Wednesday"
    : CASE 4
    :    PRINT "Thursday"
    : CASE 5
    : CASE 6
    :    PRINT "Saturday"
    : CASE 7
    :    PRINT "Sunday"
    : CASE
    :     ; default, always the last CASE statement
    :    PRINT "There are only seven days in a week!"
    : END SWITCH
```

Compatibility Issues:

This statement is only supported with Release IV or greater.

References:

SWITCH Numeric
Section 4.11 of the Programmer's Guide

CASE String

General Form:

```
CASE string-case[,string-case]...
```

Where:

```
string-case =  {alpha-variable}
               {literal-string}
```

Discussion:

This statement may only occur within a string SWITCH structure. Refer to SWITCH String for an explanation of how this statement may be used within a string SWITCH structure to define the action to take for specific case values.

Examples:

An example of valid syntax is shown below.

```
0010 CASE "Alligators"
0010 CASE Widget_Type$
0010 CASE Activity_Code$(Index)
```

An example of practical usage is shown below.

```
0010
    : SWITCH Widget_Type$
    : CASE "Gizmos","GIZMOS"
    :    PRINT 0
    : CASE "Thingammies","THINGAMMIES"
    :    PRINT 1
    : CASE
    :    PRINT "Eh?"
    : END SWITCH
```

The following is an example of default string CASE:

```
0010  SWITCH char$
   : CASE "A", "B", "C"
   :   PRINT "One of the first three letters of the alphabet."
   : CASE
   :    ; default, always the last CASE statement
   :   PRINT "One of the letters of the alphabet excluding A, B or C."
   : END SWITCH
```

Compatibility Issues:

This statement is only supported with Release IV or greater.

CASE String (cont.)

References:

SWITCH String
Section 4.11 of the NPL Programmer's Guide

CLEAR

General Form:

```
    CLEAR [V                               ]
          [N                               ]
          [P [line-number1][,[line-number2] ]]
```

Discussion:

The CLEAR command is used to clear program text and variables from user memory.

A CLEAR command with no parameters:

- Resets the current LIST module to the RUN module.
- Clears all program text from the RUN module.
- Clears all static variables from the RUN module.
- Deletes any other modules which are no longer referenced and do not have common variables defined.
- Resets the device table to default values, and turns off STEP Mode and TRACE Mode. In addition, it clears the screen and the system message appears.

The CLEAR function refers to program text in the current list module. This is set to the currently executing module whenever a program HALTs or continues, or when it is changed using the MODULE command, and can be referenced using LIST DT.

A CLEAR N or a CLEAR V command clears all static and non-common variables from the current LIST module only.

A CLEAR P command clears all program text from the current LIST module only.

CLEAR (cont.)

NOTE: Any command which can remove variable declarations or program text from a module causes the module to deresolve.

Deresolution of a module always destroys the execution return stack and removes all recursive variables, and PUBLIC variables declared by the module. Static variables are not removed immediately, but (non-common) static variables are removed before the module is reresolved.

In addition, any modules which INCLUDE the deresolved module are also deresolved.

These range parameters operate as follows:

- If line-number1 is specified, all program lines starting at line-number1 up to the end of the program are removed (e.g., CLEAR 1020).
- If ,(comma)line-number2 is specified, all program lines starting at the beginning of the program up to and including line-number2 are removed (e.g., CLEAR ,1060).
- If line-number1,line-number2 is specified, all program lines within the limits of line-number1 to line-number2 inclusive are removed (e.g., CLEAR 1020, 1060).

After executing a CLEAR statement of any kind as a programmable statement, the program terminates and may not be CONTINUEd.

CLEAR is executable only in the interpretive version of the RunTime. In the non-interpretive version, execution of a CLEAR statement of any kind causes an exit from the RunTime program.

CLEAR (cont.)

Examples:

```
:0010 PRINT "ABC"
:0020 PRINT "123"
:0030 PRINT "TEST"
:0040 FOR I=1 TO 10
    : PRINT I
    : NEXT I
:0050 J$="Y"
:0060 IF X$=J$ THEN 200
:0070 K=3
    : M=2

:CLEAR P30,50      ... would remove lines 30 through 50
:CLEAR P50         ... would remove lines 50 and greater
:CLEAR P,30        ... would remove lines up to and including 30
```

Compatibility Issues:

With NPL Revision 4.0, a CLEAR statement, with no parameters, acts upon INCLUDEd modules by effectively deleting them, if the modules are discardable under normal rules.

NOTE: This may cause /EXIT procedures to be executed in the affected modules.

The CLEAR statement is implemented in Revision 2.00 and greater of NPL.

The CLEAR statement is not a programmable statement in Wang 2200 Basic-2.

References:

Program Loading - Section 5.3 of the Programmer's Guide
Modules - Section 4.10 of the Programmer's Guide

$CLOSE

General Form:

```
     $CLOSE[{file-number   } [,{file-number   }] ...]
           [{device-address} [ {device-address}] ...]
           [<address-var>,]
```

NOTE: The use of this statement is not recommended. Use the Niakwa Data Manager as a better alternative.

Discussion:

The $CLOSE statement releases one or more devices that may have been reserved for exclusive access ("hogged") with the $OPEN statement. If no device address or file number is specified, all devices currently "hogged" by a user are released.

Examples:

```
0010 $CLOSE#1,#2,#A
```

The result of the above is that files number 1,2,A (value of variable A) are released for general use by other users.

```
0010 $CLOSE
```

The result of the above is that all files or devices "hogged" by user are released for general use by other users.

```
0010 $CLOSE/D12
```

The result of the above is that disk device D12 is released for general use by other users.

```
0010 $CLOSE/215
```

The result of the above is that printer device 215 is released for general use by other users.

```
0010 $CLOSE <A$>
```

The result of the above is that the device address stored in A$ is released for general use by other users.

$CLOSE (cont.)

Compatibility Issues:

In a multi-user environment, "device-hogging" emulates the Wang 2200 except in the case of disk devices. In this case, "hogging" on a disk address is PLATTER-specific, not DEVICE-specific.

For example, on a Wang 2200, assuming D20 is the removable disk on a Wang 2200 phoenix drive, $OPEN /D20 "hogs" the entire device (disks /D20 - /D25); whereas, under NPL, $OPEN /D20 "hogs" only disk /D20.

Use of the address-var parameter is supported only on NPL Revision 3.0 or greater and is not supported on the Wang 2200.

References:

$OPEN
Exclusive Access - Section 7.2.4 of the Programmer's Guide

COM

General Form:

```
     COM variable[,variable]...
```

Where:

```
     variable    = {numeric-scalar                       }
                   {numeric-array-name(sub1[,sub2]}      }
                   {alpha-array-name(sub1[,sub2])[length]}
                   {alpha-scalar[length]                 }

     dim1, dim2 = numeric-expression (1 ≤ dim ≤ 65535)

     length      = positive integer or numeric-scalar-variable
                   such that 1 ≤ length ≤  65535
```

Discussion:

The COM statement is used to reserve memory for variables whose values should not be cleared whenever new program overlays are loaded into memory. Variables defined in a COM statement remain in memory when a program is cleared or overlayed by a new program, unlike non-common variables (defined by DIM) which are cleared when a new program is loaded or the current program is cleared. Common variables are useful in passing data through multiple program overlays.

The COM statement must appear before any non-common variables are defined by DIM statements or by reference. If a common variable is to be used by more than one program running sequentially, the COM statement needs only to declare it in the first program. An error is not generated if it appears in a COM statement in later programs, provided it is given the same dimensions as in earlier programs.

Common variables are cleared by the CLEAR command and the CLEARV command, and any time a LOAD RUN statement is executed.

Variables declared as COM are always declared as static variables which are private to the module. They are never recursive, PUBLIC or local to a function, even if they appear where a DIM statement would give the declared variable these attributes (i.e., in a PUBLIC section or within the body of a function).

COM (cont.)

Use of COM variables to mark library modules as resident.

Normally, a library module (one that has been loaded as a result of an INCLUDE statement) cannot perform overlays and has no need for COM type variables. Variables in a library module are not affected by programmed overlays, which affect only the RUN module.

When a library module is no longer referenced by INCLUDE statements, both the code and variables associated with it are removed. This occurs after the RUN module has been resolved, but before it starts execution.

However, if the library module declares any common (COM) variables, NPL will not delete the module. In this case, any COM variable serves as an indicator that the module should be retained for future use.

A library that uses COM variables in this way should provide some public function which will execute a COM CLEAR statement, in order that the module may be explicitly deleted when it is no longer wanted.

Either dimension of an array may be specified as containing up to 65535 elements and the length of any variable may be specified up to 65535 bytes. However, the maximum total number of array elements must not exceed 65535 and the total size of the array (number of elements * length) must not exceed 65535 bytes.

Dynamic Variable Dimensioning with COM

The length of variables defined by a COM statement can be specified by a numeric-scalar-variable that has been defined as a COMmon variable and assigned a value in a prior program.

The dimensions of a variable in the COM statement are permitted to be a numeric-expression. However, if variables are used in the expression, they must be declared in a previous COM statement. Useful expressions normally use as terms only constants, common variables and, possibly, the SPACE function in some combination.

Default variable declaration is assumed by NPL when a variable reference appears without any previous explicit declaration. Here, NPL takes one of two actions:

COM (cont.)

a. If $OPTION byte 38 is set to HEX(01), an error occurs. All variable references must be preceded by a declaration.

b. If $OPTION byte 38 is set to HEX(00), a variable may be declared by default in some cases, depending on the context in which the first variable reference in the program appears, according to the following table:

Location of First Variable Reference	**Default Allocation Types**
Within a function body	Not legal--error occurs
Outside all function bodies	DIM/STATIC

NOTE: Constant variables must always be explicitly declared.

Examples:

```
0010 COM A$(SPACE-20000)1, B$((SPACE-1000)/C2)C2,C$(MAX(256,J))1
```

Declares variables which use, respectively, all but the last 20K of memory, the remaining memory (after A$()) less 1K set up as elements of C2 bytes each (assuming C2 is a common variable), and an array with either J bytes or 256 bytes, whichever is larger (again, J is assumed to be a common variable).

```
0010 COM X$24, Q$(4)4, X(4,4)
```

```
PROGRAM1:

 :0010 COM A,B,C
 :0020 A=10: B=20: C=4
 :0030 LOAD T"PROGRAM2"
 :RUN

PROGRAM2:

 :0010 COM X$(A,B)C
```

This defines an array of 10 by 20 elements, each 4 bytes in length.

COM (cont.)

Compatibility Issues:

The Wang 2200 Basic-2 limitation of 124 characters on the length of a scalar has been extended to 65535. (Be aware, however, that 124 bytes is still the largest scalar variable length which can be saved using a DATASAVE DC statement).

The memory overhead for variables is greater under NPL than on a Wang 2200:

			Overhead under NPL	
e.g.	Variable Type	**Overhead on Wang 2200**	**One dimension**	**Two dimension**
X	Numeric Scalar	4 bytes	8 bytes	N/A
X()	Numeric Array	6	12 bytes	14 bytes
X$	Alpha Scalar	5	10 bytes	N/A
	Alpha Array	7	14 bytes	16 bytes

Wang 2200 Basic-2 does not allow dimensions in a COM statement to be defined as an expression. The Wang 2200 allows only constant and numeric-scalar common variables to be used as variable dimensions.

References:

CLEAR[V]
COM CLEAR
DIM
LOAD RUN

COM CLEAR

General Form:

```
COM CLEAR [variable-name]
```

NOTE: The use of this statement is not recommended. Use program modules as a better alternative.

Discussion:

The COM CLEAR statement is used to change the status of variables from common to non-common or from non-common to common. The dimensions and values of the variables are not changed.

If no variable is specified in the COM CLEAR statement, all common variables are redefined as non-common variables.

If a common variable name is specified in the COM CLEAR statement, that variable and all variables that appeared in a COM statement after it are changed to non-common variables (as if they were specified in a DIM statement), while all common variables defined before the specified variable remain common.

If a non-common variable is specified in the COM CLEAR statement, all non-common variables defined before the specified variable become common variables.

COM CLEAR (cont.)

Examples:

```
0010 COM CLEAR
0010 COM CLEAR Q9$
0010 COM CLEAR X$()

:0010 COM A$,B$
:0020 DIM C$,D$
:0030 COM CLEAR D$ :REM A$, B$, and C$ are now common variables
                        D$ is non-common
:0040 COM CLEAR B$ :REM B$,C$, and D$ are now non-common variables
                        A$ is still common
:RUN
```

Compatibility Issues:

References:

COM
DIM

& (Concatenation) Alpha-Operator

General Form:

```
     alpha-receiver = alpha-operand [ & alpha-operand ]...
```

Where:

```
     alpha-operand = {literal-string }
                     {alpha-variable }
                     {ALL function   }
                     {BIN function   }
                     {system-variable}
```

Discussion:

The concatenation alpha-operator combines the contents of the first alpha-operand with the contents of the second alpha-operand without intervening characters into a single character string and assigns the result to the alpha-receiver.

The concatenation alpha-operator may only be used in an alpha-expression in an alpha-assignment statement. Further, the concatenation alpha-operator is treated specially and may not be combined with any other alpha-operator (except itself) in the same alpha-expression.

Examples:

```
:0010 A$="ONE" & "TWO"
:0020 PRINT A$
:RUN
ONETWO

:0010 A$="ONE" & HEX(2B) & "TWO"
:0020 PRINT A$
:RUN
ONE+TWO
```

Compatibility Issues:

References:

LET Alpha-assignment

CONTINUE

General Form:

```
CONTINUE
```

Discussion:

The CONTINUE command is used to resume normal execution of a program which has temporarily been halted and is in Immediate Mode.

If a program is unresolved and the CONTINUE command is entered, an error occurs (ERR A09 - Program Not Resolved).

When executed as a program statement, the CONTINUE statement causes STEP Mode to be exited if in effect.

The CONTINUE statement performs no operation on the non-interpretive RunTime program.

Examples:

```
:CONTINUE
:PRINT X: CONTINUE
100 CONTINUE
```

Compatibility Issues:

This statement is only supported with Release 2.00 or greater.

The CONTINUE statement is not a programmable statement in Wang 2200 Basic-2.

References:

STEP
Exiting Immediate Mode - Section 2.6.4 of the Programmer's Guide

CONTINUE LOAD

General Form:

```
CONTINUE LOAD
```

Discussion:

The CONTINUE LOAD command is used to resume normal execution of a program which has temporarily been halted and is in Immediate Mode, until the next LOAD statement is completed. At this point, Immediate Mode is reactivated. CONTINUE LOAD is primarily a debugging tool which allows specific program modules to be inspected without stepping through entire programs.

If a program is unresolved and the CONTINUE LOAD command is entered, an error occurs (ERR A09 - Program Not Resolved).

When executed as a program statement, the CONTINUE LOAD statement causes STEP Mode to be exited, if in effect, until the next LOAD statement is executed.

The CONTINUE LOAD statement performs no operation on the non-interpretive Run-Time program.

Examples:

```
:CONTINUE LOAD
0010 PRINT A$: CONTINUE LOAD
```

Compatibility Issues:

This statement is only supported with Release 2.00 or greater.

This statement is not valid in Wang 2200 Basic-2.

References:

LOAD
STEP
Inspection and Modification of Program Logic - Section 6.3 of the Programmer's Guide

CONTINUE NEXT

General Form:

```
CONTINUE NEXT
```

Discussion:

The CONTINUE NEXT command is used to temporarily resume normal execution of a program which is in Immediate Mode (assuming the program was halted during normal execution). The program continues normally until the loop associated with the most recently executed FOR statement is completed. At this point, program execution again halts, allowing for further Immediate Mode command entry. This is primarily a debugging tool which allows FOR/NEXT loops to be rapidly completed while single-stepping through a program.

If a program is unresolved and the CONTINUE NEXT command is entered, an error occurs (ERR A09 - Program Not Resolved).

If program execution is not in a FOR/TO loop, CONTINUE NEXT generates an error (P40 - No Corresponding FOR for NEXT Statement).

The FOR/TO loop may have been initialized using either the unstructured FOR/TO statement or the FOR/BEGIN structured statement. It may not be used to rapidly complete other types of structured loops, such as WHILE/WEND or REPEAT/UNTIL loops.

NOTE: **The CONTINUE NEXT statement works only with the highest level of the return stack. Therefore, execution of a CONTINUE NEXT statement is invalid when in a subroutine called from within a loop. If subsequent NEXT statements are encountered after the CONTINUE NEXT statement is executed, program execution continues unaffected. Only completion of the corresponding FOR/TO loop causes Immediate Mode to be invoked. The LIST STACK command can be used to examine the current status of active FOR/NEXT loops and subroutines.**

The CONTINUE NEXT statement is legal wherever a NEXT statement is legal.

The CONTINUE NEXT statement performs no operation on the non-interpretive Run-Time program.

CONTINUE NEXT (cont.)

Examples:

```
CONTINUE NEXT

:0010 FOR X=1 TO 100
:0020    PRINT "X= ";X
:0030    FOR Y=1 TO 100
:0040       PRINT "Y= ";Y
:0050    NEXT Y
:0060    PRINT "THIS IS IN THE "X" LOOP"
:0070 NEXT X
:0080 PRINT "BOTH LOOPS COMPLETE"
```

In this example, assume that STEP Mode has been invoked. If a CONTINUE NEXT statement is entered before program execution of the FOR Y statement at line 30, program execution continues until line 80 (until the FOR X loop is completed). If a CONTINUE NEXT statement is entered during execution of the FOR Y loop (at lines 40 or 50), program execution continues only until the FOR Y loop is completed.

Compatibility Issues:

This statement is only supported with Release 2.00 or greater.

This statement is not valid in Wang 2200 Basic-2.

References:

FOR/TO
LIST STACK
NEXT
Inspection and Modification of Program Logic - Section 6.3 of the Programmer's Guide

CONTINUE RETURN

General Form:

```
CONTINUE RETURN
```

Discussion:

The CONTINUE RETURN command is used to temporarily resume normal execution of a program which is in Immediate Mode. The program continues normally until the RETURN statement associated with the most recently executed GOSUB statement is encountered. At this point, program execution again halts, allowing for further Immediate Mode command entry. This is primarily a debugging tool which allows subroutines to be rapidly completed while single-stepping through a program.

If a program is unresolved and the CONTINUE RETURN command is entered, an error occurs (ERR A09 - Program Not Resolved).

If program execution is not in a subroutine, CONTINUE RETURN generates an error (P41 - RETURN without GOSUB).

NOTE: **The CONTINUE RETURN command works only with the highest level of the return stack. However, execution of a CONTINUE RETURN statement is valid when in a loop initialized in a subroutine. In addition, any subsequent subroutines which are encountered during execution of a CONTINUE RETURN executes fully without interruption (the RETURN statement associated with the subsequent subroutine does not halt the program). Program operation continues until the RETURN statement associated with the subroutine most recently executed at the time the CONTINUE RETURN command is encountered.**

The CONTINUE RETURN statement is legal whenever a RETURN statement is legal. CONTINUE RETURN may also be used to rapidly complete the current PROCEDURE or FUNCTION call, when no additional GOSUB statements are pending since the PROCEDURE/FUNCTION call was made.

In the case of a PROCEDURE, execution halts at the statement following the statement containing the most recently called PROCEDURE.

CONTINUE RETURN (cont.)

In the case of a FUNCTION, execution halts as soon as possible after returning the value of the function, and before executing any additional statements. In the most common case, statements will only contain one function call; in that case, execution will be halted at the following statement.

For example:

```
PRINT ’get_number : PRINT "next statement"
```

In the above, the results from the FUNCTION ’get_number will be printed and execution will stop before the phrase "next statement" is printed.

If the function is used as a term in a statement containing additional function calls, then execution halts at the first statement within the body of the next function call.

For example:

```
PRINT STR( string$, ’start_index, ’number_chars )
```

If execution in the above were halted right after returning the value from the FUNCTION ’start_index, then the next statement to be executed would be the first statement encountered within the FUNCTION ’number_chars.

Examples:

```
:0010 GOSUB ’100
:0020 PRINT "After completion of ’100"
:0030 END
:0500   DEFFN’100
:0510     PRINT "In ’100"
:0520     GOSUB ’200
:0530     PRINT "After completion of ’200"
:0540   RETURN
:0700       DEFFN’200
:0710       PRINT "In ’200"
:0720   RETURN
:RUN
```

CONTINUE RETURN (cont.)

In this example, assume that STEP Mode has been invoked. If a CONTINUE RETURN statement is entered (in Immediate Mode) before execution of the PRINT statement on line 510, normal program execution continues until both subroutines ’100 and ’200 are complete. Program execution halts (and Immediate Mode becomes available) before execution of the PRINT statement on line 20. If a CONTINUE RETURN statement is entered before execution of the PRINT statement on line 710, normal program execution continues only through the completion of subroutine ’200. Program execution halts (and Immediate Mode becomes available) before execution of the PRINT statement on line 530.

CONTINUE RETURN is also permitted if the top function call is a call to a PROCEDURE or FUNCTION. In the case of a PROCEDURE, execution halts when the procedure exits using RETURN or executes END PROCEDURE. A FUNCTION call halts at the start of the next statement following the RETURN (value).

NOTE: If the FUNCTION result is a parameter to another FUNCTION, this may not be the statement following the one in which the FUNCTION was called.

For example:

```
:list
0010 FUNCTION ’Stuff(X)
: RETURN (X)
: END FUNCTION
0020 FUNCTION ’Nonsense(T)
:STOP #
: RETURN (T)
: END FUNCTION
0030 PRINT ’Stuff(’Nonsense(3))
0040 PRINT "Done"
:run

STOP 0020
:list stack
0030 PRINT ’Stuff(’Nonsense(3))
’Nonsense
:continue return
’Stuff 0010 : RETURN (X): END FUNCTION
```

Compatibility Issues:

This statement is only supported with Release 2.00 or greater.

CONTINUE RETURN (cont.)

The CONTINUE RETURN statement performs no operation on the non-interpretive Run-Time.

This statement is not valid in Wang 2200 Basic-2.

References:

END PROCEDURE
GOSUB
RETURN
STEP
Inspection and Modification of Program Logic - Section 6.3 of the Programmer's Guide

CONVERT

General Form:

```
CONVERT alpha-variable TO numeric-receiver

CONVERT numeric-expression TO alpha-variable,(image)
```

Where:

```
image = {[+] [$] [#[,]]...[.][#]...[^^^^] [+ ]}
         [-]                              [- ]
                                          [++]
                                          [--]
        {alpha-variable containing image     }
```

Discussion:

The CONVERT statement is used to convert an alpha-variable to a numeric-variable or a numeric-expression to an alpha-variable.

Alpha-variable to Numeric-receiver:

This form of the CONVERT statement is used to convert the contents of an alpha-variable to a numeric value. In this case, an image format to receive the numeric values is not required.

The contents of the alpha-variable specified must be an ASCII representation of a valid numeric value. Otherwise, an error X75 (Illegal Number) generated.

Only the following characters are allowed in the alpha-variable: digits from "0" to "9", "+" and "-" signs, decimal points (".") and spaces. Dollar signs ("$"), trailing signs, "DB" and "CR", and commas cannot be part of the alpha-variable.

Numeric-expression to Alpha-variable:

This form of the CONVERT statement is used to convert the numeric value of the specified numeric-expression to an alpha-string in a format specified by image, and store the results in the specified alpha-variable. The alpha-variable must be dimensioned large enough to hold the alpha-string, or only the first characters of the result are stored.

CONVERT (cont.)

The image specified must conform to the following rules:

1. A minimum of one "#" is required in the image.

2. Both leading and trailing signs may not occur within the same image.

3. The maximum number of characters allowed in an image is 254.

Two general formats of the image specification are supported: fixed point (###.##) and exponential (##.###^^^^). The image controls the format of the alpha-variable based on the numeric-expression as follows:

1. Each number sign ("#") corresponds to one digit.

2. For fixed-point images, if the number of digits to the left of the decimal point in the numeric-expression exceeds the number of number signs ("#") to the left of the decimal point in the image, an error X71 (Value Exceeds Format) occurs. If the image exceeds the expression, leading zeros are added. If the number of digits to the right of the decimal point exceeds the number of number signs ("#") to the right of the decimal point in the image, any extra digits are truncated.

3. The comma (","), and decimal point (".") are inserted at the proper location.

4. Sign ("+" or "-") may be specified as either the first or last byte of the image. If minus ("-") is used, the sign is only generated for negative values, a blank is inserted for positive values. If a plus ("+") is used, the real sign of the value is always generated. If no sign is specified, the absolute value of the expression is converted and no sign byte is present in the alpha-variable.

5. The dollar sign ("$") is placed in the image at the corresponding location. It is either the first byte or the second byte (if a leading sign is specified) of the alpha-variable.

6. Any spaces within the image are ignored.

CONVERT (cont.)

7. If the image format ends with a "++" or "--" sign, a "CR" or "DB", respectively, is placed at the end of the alpha-variable string for any negative values. Positive values end with two spaces.

8. If exponential format (^^^^) is specified, the value of the expression is converted to the exponential format. In this case, the value is scaled to the format specification and the exponent is set accordingly. The exponent is expressed in the form "E±nn" where nn is the exponent.

Examples:

```
0010 CONVERT X$ TO X                        :REM ALPHA TO NUMERIC
0010 CONVERT STR(Q1$(),22,6) TO X1          :REM ALPHA TO NUMERIC
0010 CONVERT X TO X1$,(#####.##)            :REM NUMERIC TO ALPHA
0010 CONVERT X(4)TO STR(Q$,14,9),(-#,###.##):REM NUMERIC TO ALPHA
0010 CONVERT A(I) TO B$,(##.##^^^^)         :REM NUMERIC TO ALPHA
0010 CONVERT A(I) TO B$,(C$)                :REM NUMERIC TO ALPHA

:0010 X=123.4567
:0020 CONVERT ROUND (-X,2) TO A$,($#,###.##++)
:0030 CONVERT X TO B$,(###)
:0040 CONVERT X TO C$,(##.#^^^^)
:0050 F$="+###.##"
    : CONVERT X TO D$,(F$)
:0060 PRINT A$,B$,C$,D$
:RUN
$0,123.46CR     123             12.3E+01        +123.45
```

Compatibility Issues:

References:

COPY

General Form:

```
     COPY T [file#,         ]  [(start-sector,end-sector)]
            [disk-address,  ]
            [<address-var>, ]

       TO T [file#,         ]  [(destination-sector)     ]
            [disk-address,  ]
            [<address-var>, ]
```

Where:

```
     start-sector       = numeric-expression

     end-sector         = numeric-expression

     destination-sector = numeric-expression
```

NOTE: This statement is supported for Wang compatibility reasons only and its use in new development is not recommended.

Discussion:

The COPY statement is used to copy sectors from the first diskimage to the second diskimage. The information to be copied may be a range of sectors, specified in parentheses after the originating disk address, and may be copied starting at a certain sector on the destination diskimage, specified in parentheses after the destination disk address. If no destination sector is specified, the same starting sectors on both diskimages are used. If no start-sector or end-sector is specified for the originating diskimage, all sectors up to the current catalog end are copied.

Examples:

```
0010 COPY T/D20, TO T/D22,
```

Copy from D20 to D22 from sector 0 to the current end of catalog of D20.

```
0010 COPY T/D21,(10,300) TO T
```

Copy from D21, sectors 10 through 300 to currently selected disk address, starting at sector 10.

COPY (cont.)

```
0010 COPY T(0,5000) TO T#2,(6000)
```

Copy from current disk address sectors 0 through 5000 to the disk specified as file#2 in the internal device table, starting at sector 6000.

```
0010 COPY T#X,(300,400) TO T#Y,
```

Copy from the disk specified as file #X, sectors 300 through 400, to the disk specified as file #Y in the internal device table, starting at sector 300.

```
0010 COPY T<A$>, TO T<B$>,
```

Copy from the device address stored in <A$> to the device address stored in <B$>, from sector 0 to the current end of catalog of <A$>.

Compatibility Issues:

Use of the address-var parameter is supported only on NPL Revision 3.0 or greater and is not supported on the Wang 2200.

The COPY statement executes as it does on the Wang 2200, except when the results of the COPY would cause data to be written beyond the physical end of the diskimage.

On a Wang 2200, an error I98 (Illegal Sector Address or Plotter Not Mounted) would result.

In NPL, the destination diskimage would be extended to the size needed to accommodate the write operation. If insufficient disk space is available, an error I98 (Illegal Sector Address or Plotter Not Mounted) would result.

Any new space allocated would be non-cataloged disk space.

Refer to Section 7.3.4 of the Programmer's Guide for details on dynamic size of disk -image files.

NOTE: **The use of non-cataloged disk space is not recommended. Refer to Chapter 5 of the appropriate NPL Supplement for details on support of non-cataloged disk space on the hardware system.**

COPY (cont.)

References:

Internal Device Table - Section 7.2.3 of the Programmer's Guide
Dynamic Allocation - Section 7.3.4 of the Programmer's Guide
Direct Access - Section 7.3.9 of the Programmer's Guide
Extended Diskimages - Section 7.3.10 of the Programmer's Guide

COS Function

General Form:

COS*(numeric-expression)*

Discussion:

The COS function computes the value of the cosine of a numeric-expression. This is valid wherever a numeric expression is legal.

The calculation is performed in Degrees, Radians, or Gradians, depending on last execution of SELECT [D,R,G] statement.

Examples:

```
0010 B = Y4+W6+32*COS(E9-10)
0010 H(K) = M3-COS(N-INT(N/90)*90)
0010 V7 = 25
0020 X8 = COS (V7 + 45)
0030 PRINT X8
:RUN
:.63331920308472
```

Compatibility Issues:

Due to the use of a different algorithm, results of these functions may differ from functions evaluated on a Wang 2200. In general, however, the functions are accurate to 13 significant digits.

References:

SELECT [D,R,G]

DAC Alpha-operator

General Form:

```
alpha-receiver = [...] DAC alpha-operand [...]
```

Where:

```
alpha-operand = {literal-string  }
                {alpha-variable  }
                {ALL function    }
                {BIN function    }
                {system-variable }
```

Discussion:

The DAC (decimal add with carry) alpha-operator adds the decimal value of the alpha-operand to the decimal value of the alpha-receiver. The DAC alpha-operator may only be used in an alpha-expression in an alpha-assignment statement.

The DAC operation assumes that both operands contain valid, unsigned BCD (Binary Coded Decimal) data, where data consists of two digits per byte, and each of those digits is a number between 0 to 9. DAC does not check the operand contents for validity prior to adding; consequently, the resultant is unpredictable if operands contain invalid data.

Each byte of the alpha-operand is added (in base 10 arithmetic) to each corresponding byte of the receiving alpha-variable; carry propagation is automatically performed between bytes.

If the values of the alpha-operand and the receiving alpha-variable are of different length, then the DAC algorithm implicitly extends the shorter value with leading zeroes prior to the operation. If the resultant is larger than the receiving alpha-variable, then the extraneous high order bytes of the resultant are truncated before assignment.

NOTE: Contrary to conventional alpha-variable operations, the DAC alpha-operator operates on all bytes of an alpha-variable (either as a receiver or an alpha-operand), including trailing spaces.

DAC Alpha-operator (cont.)

Examples:

```
0010 A$=DAC MyRec$.field1$
0010 A$=B$ DAC HEX(0001)
0010 A$=DAC STR(B$,5,3)

:0010 DIM A$3,C$3
:0020 PACK(######) A$ FROM 9990
:0030 C$=A$ DAC HEX(0060)
:0040 PRINT HEXOF(C$)
:RUN
10050
```

Compatibility Issues:

The Decimal Add with Carry operation accepts invalid packed decimal numbers as an alpha-expression in Wang 2200 Basic-2. In this case, the results are predictable but meaningless.

NPL is compatible with Wang 2200 Basic-2 with respect to the DAC operator, provided the alpha-expression contains valid packed decimal values.

References:

DSC
PACK
$PACK
UNPACK
$UNPACK
VER

DATA

General Form:

```
DATA {literal-string  }[,{literal-string  }]...
     {numeric-constant}{numeric-constant  }
```

Discussion:

The DATA statement is used to define a list of alpha or numeric constants that are used as input to the READ and MAT READ statements.

The DATA statement provides a method of storing tables of alpha and/or numeric constants within a program. The RESTORE statement allows repetitive use of the DATA statement values by resetting DATA pointers to a specified point.

The DATA statement is not valid in Immediate Mode.

Examples:

```
0010 DATA 6
0010 DATA "ABC", HEX(10),HEX(20)
0010 DATA 6, 9, 8, 4, 40, 35.6, 3E06
0010 DATA "ABC", "COMPANY 7", "MARCH 25, 1983"
0010 DATA 1, "A", 2, "B", 3E06, "Z"
```

Compatibility Issues:

References:

MAT READ
READ
RESTORE

DATA LOAD BA

General Form:

```
    DATALOAD BA T [file-number,  ] (expr1[,return-value])
                  [disk-address, ]
                  [<address-var>,]
                                                      alpha-variable
```

Where:

```
    expr1          = an alpha-variable or numeric-expression.

    return-value   = an alpha-variable or numeric-receiver.

    alpha-variable = alpha-variable into which the data is to
                     be loaded (must be ≥ 256 bytes).
```

NOTE: The use of this statement is not recommended. Refer to the Niakwa Data Manager as a better alternative.

Discussion:

The DATA LOAD BA statement is used to load the raw, unformatted contents of a specified sector address (expr1) into the first 256 bytes of the specified alpha-variable. If the alpha-variable is dimensioned larger than 256 bytes, the remaining bytes are not affected by the DATALOAD BA statement.

Expr1 contains the sector-number to be loaded. If expr1 is an alpha-variable, the binary value of the first two bytes is used.

After execution of the statement, the return-value contains the sector-number immediately following the sector number accessed by the operation. If return-value is an alpha-variable, the value is contained in the first two bytes in binary.

Use of an alpha-variable to contain sector addresses results in improper sectors being accessed if extended (greater than 16 MB) diskimages are in use and the sector numbers being accessed are greater than 65535. Refer to Section 7.3.10 of the Programmer's Guide for additional programming considerations for use of extended diskimages.

DATA LOAD BA (cont.)

DATALOAD BA is a direct access instruction, as opposed to a catalog instruction. That is, the internal device table is not used or affected by a DATALOAD BA instruction (except to determine the diskimage address if a file-number is specified).

Use of FUNCTIONs as arguments in a DATALOAD or DATASAVE statement, which does not have the platter $OPEN, may result in a disk operation which is not integral. Any implied lock that RTI would issue against the disk is released for the duration of the function call. If the application depends on this implied lock to maintain data integrity, arguments should be evaluated separately from the DATALOAD/DATASAVE statement.

For example:

```
Dim Array1$(22)12,Array2$(22)12

DATA SAVE DA T#1,(X,X)Array2$(),’FunctionResult$,array2$()
DATA LOAD DA T#1,(X,X)Array1$(),Key$(’NextIndex),Array2$()
```

HINT: Function calls should not be used when evaluating arguments in DATALOAD / DATASAVE statements.

Examples:

```
0010 DATA LOAD BA T(Q$,Q$) A$()
0010 DATA LOAD BA T#1, (X-1,X) X$
0010 DATA LOAD BA T#S, (R$,R) STR(Z$(),257,256)
0010 DATA LOAD BA T/D10, (R$) Z$
0010 DATA LOAD BA T#4, (X) K$()
0010 DATA LOAD BA T/D12, (30) R$()
0010 DATA LOAD BA T<A$>, (30) R$()
```

Compatibility Issues:

Wang 2200 Basic-2 requires that the receiving variable be an array-variable. NPL allows an array-variable **or** an alpha-scalar variable as the receiving variable.

Use of the address-var parameter is only supported by NPL Revision 3.0 or greater and is not supported on the Wang 2200.

References:

DATA SAVE
Direct Access - Section 7.3.9 of the Programmer’s Guide
Extended Diskimages - Section 7.3.10 of the Programmer’s Guide

DATA LOAD BM

General Form:

```
     DATALOAD BM T [file-number,  ] (expr1[,return-value])
                   [disk-address, ]
                   [<address-var>,]
                                                alpha-variable
```

Where:

```
     expr1          = an alpha-variable or numeric-expression.

     return-value   = an alpha-variable or numeric-receiver.

     alpha-variable = alpha-variable into which data is to be
                      loaded.
```

NOTE: The use of this statement is not recommended. Refer to Niakwa Data Manager as a better alternative.

Discussion:

The DATALOAD BM statement is used to load the raw, unformatted contents of the specified sector address (expr1) into the specified alpha-variable. Enough sectors are read to fill the specified alpha-variable. A buffer of zero bytes reads zero sectors. If the operation would require that a sector beyond the physical end of the diskimage be read, an I98 error (Illegal Sector Address and Platform Not Mounted) results and the data in the specified alpha-variable is undefined.

Expr1 contains the starting sector-number to be loaded. If expr1 is an alpha-variable, the binary value of the first two bytes is used.

NOTE: Use of alpha-variables for the starting sector-number is not advised for diskimage files where the EXT=Y clause has been specified. Refer to Section 7.3.10 of the Programmer's Guide for further details on extended diskimages.

After execution of the statement, the return-value contains the sector-number immediately following the last sector number accessed by the operation. If return-value is an alpha-variable, the value is contained in the first two bytes in binary.

DATA LOAD BM (cont.)

NOTE: An error P51 (Variable or Value Too Short) results if an alpha-variable is specified as the return-variable and the sector number exceeds 65535 (extended diskimage in use). Refer to Section 7.3.10 of the Programmer's Guide for further details on extended diskimages.

DATALOAD BM is a direct access instruction as opposed to a catalog instruction; therefore, the internal device table is not modified by a DATALOAD BM instruction.

Use of FUNCTIONs as arguments in a DATALOAD or DATASAVE statement, which does not have the platter $OPEN, may result in a disk operation which is not integral. Any implied lock that RTI would issue against the disk is released for the duration of the function call. If the application depends on this implied lock to maintain data integrity, arguments should be evaluated separately from the DATALOAD/DATASAVE statement.

For example:

```
Dim Array1$(22)12,Array2$(22)12

DATA SAVE DA T#1,(X,X)Array2$(),'FunctionResult$,Array2$()
DATA LOAD DA T#1,(X,X)Array1$(),Key$('NextIndex),Array2$()
```

HINT: Function calls should not be used when evaluating arguments in DATALOAD / DATASAVE statements.

Examples:

```
0010 DATA LOAD BM T(Q$,Q$) A$()
0010 DATA LOAD BM T#1, (X-1,X) X$()
0010 DATA LOAD BM T#S, (R$,R) STR(Z$(),513,512)
0010 DATA LOAD BM T<A$>, (R$) Z$()
0010 DATA LOAD BM T/D12, (30) R$()
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

On a Wang 2200, the highest sector number that can be referenced is 65535. In NPL, if the EXT=Y clause is specified on the $DEVICE statement, sector numbers above 65535 can be used. Refer to Section 7.3.10 of the Programmer's Guide for further details on extended diskimages.

Use of the address-var parameter is not supported on the Wang 2200.

DATA LOAD BM (cont.)

References:

DATA LOAD BA
DATA SAVE BM
Direct Access - Section 7.3.9 of the Programmer's Guide
Extended Diskimages - Section 7.3.10 of the Programmer's Guide

DATA LOAD DA

General Form:

```
DATALOAD DA T [file-number,  ] (expr1[,return-value])
              [disk-address, ]
              [<address-var>,]
                                       variable-list
```

Where:

```
expr1         = an alpha-variable or numeric-expression.

return-value  = an alpha-variable or numeric-receiver.

variable-list = {alpha-variable   }[,{alpha-variable    }]...
                {alpha-array      }{alpha-array         }
                {numeric-receiver }{numeric-receiver    }
                {numeric-array    }{numeric-array       }
```

NOTE: The use of this statement is not recommended. Refer to Niakwa Data Manager as a better alternative.

Discussion:

DATA LOAD DA is used to load logical data records beginning at a specified sector number into a specified variable-list. The variable-list may contain a mix of alpha and numeric variables.

Expr1 contains the first sector-number to be loaded. If expr1 is an alpha-variable, the binary value of the first two bytes is used.

After execution of the statement, the return-value contains the sector-number immediately following the last sector number accessed by the operation. If return-value is an alpha-variable, the value is contained in the first two bytes in binary.

If an end-of-file trailer record is encountered during a read, no additional data is read, and the remaining variables in the variable-list retain their current values. The end-of-file condition is then set (and can be tested with an IF END THEN statement), and the return-value is set to the sector number of the end-of-file trailer record.

DATA LOAD DA (cont.)

Values are assigned sequentially to variables in the list. The variable list may include array designators such as A$() or A(). This indicates that the entire array is to be loaded element-by-element. An attempt to load numeric data into an alpha-variable or alpha data into a numeric-variable generates an error. If an alpha value is shorter than the variable, the variable is padded with spaces. If an alpha value is longer than the variable, the variable is filled with the truncated value.

Normally, the data have been previously saved with a DATA SAVE DC or DATA SAVE DA statement using a variable-list with the identical types and sizes of variables listed in the same order.

DATALOAD DA is a direct access instruction, as opposed to a catalog instruction; therefore, the internal device table is not used or affected by a DATALOAD DA instruction (except to determine the diskimage address if a file-number is specified).

Use of FUNCTIONs as arguments in a DATALOAD or DATASAVE statement, which does not have the platter $OPEN, may result in a disk operation which is not integral. Any implied lock that the RunTime would issue against the disk is released for the duration of the function call. If the application depends on this implied lock to maintain data integrity, arguments should be evaluated separately from the DATALOAD/DATASAVE statement.

For example:

```
Dim Array1$(22)12,Array2$(22)12

DATA SAVE DA T#1,(X,X)Array2$(),’FunctionResult$,Array2$()
DATA LOAD DA T#1,(X,X)Array1$(),Key$(’NextIndex),Array2$()
```

HINT: Function calls should not be used when evaluating arguments in DATALOAD / DATASAVE statements.

Examples:

```
0010 DATA LOAD DA T(Q$,Q$) A$,B
0010 DATA LOAD DA T#1, (X-1,X) A$(), B$, C(), D()
0010 DATA LOAD DA T#S, (R$,R) X, Y, Z$()
0010 DATA LOAD DA T/D10, (R$) Z$(), X, A$
0010 DATA LOAD DA T#4, (X) K$(), STR(A$,2,3), A(2)
0010 DATA LOAD DA T/D12, (30) R$, S$
0010 DATA LOAD DA T<A$>, (30) R$, S$
```

DATA LOAD DA (cont.)

Compatibility Issues:

This statement is supported only with Release 3.0 or greater and is not supported on the Wang 2200.

References:

DATA SAVE DA, DC
IF END THEN
Cataloged Files - Section 7.3.7 of the Programmer's Guide

DATA LOAD DC

General Form:

```
    DATALOAD DC[file-number,]variable-list
```

Where:

```
    variable-list = {alpha-variable  }[,{alpha-variable  }]...
                    {alpha-array     }  {alpha-array     }
                    {numeric-receiver}  {numeric-receiver}
                    {numeric-array   }  {numeric-array   }
```

NOTE: The use of this statement is not recommended. Refer to Niakwa Data Manager as a better alternative.

Discussion:

The DATA LOAD DC statement is used to load logical data records from a cataloged disk file into a specified list of variables. Each execution of the statement reads one or more logical data records until the variable-list is filled.

Values are assigned sequentially to variables in the list. The variable-list may include array designators such as A$() or A(). This indicates that the entire array is to be loaded element-by-element. An attempt to load numeric data into an alpha-variable or alpha data into a numeric-variable generates an error. If an alpha value is shorter than the variable, the variable is padded with spaces. If an alpha value is longer than the variable, the variable is filled with the truncated value.

If there is data beyond what can be contained in the variables-list, that data is disregarded.

If an end-of-file trailer record is encountered during a read, no additional data is read, and the remaining variables in the variable-list remain at their current values. The end-of-file condition is then set (and can be tested with an IF END THEN statement), and the current sector pointer is set to the sector address of the end-of-file trailer record.

The file-number corresponds to a file previously opened with a DATASAVE DC OPEN or DATALOAD DC OPEN statement.

DATA LOAD DC (cont.)

Normally, the data has been previously saved with a DATASAVE DC or DATASAVE DA statement, using a data list with the identical types and sizes of variables listed in the same order.

DATALOAD DC is a catalog instruction. That is, the starting sector to be used for the read is determined from the "current" slot of the internal device table entry for the file # specified. The "current" slot of the internal device table is updated to the sector# following the last sector read by the statement.

Examples:

```
0010 DATA LOAD DC A$,B
0010 DATA LOAD DC #1, A$(), B$, C(), D()
0010 DATA LOAD DC #S, X, Y, Z$()
```

Compatibility Issues:

In Revision 4.0, if the status of open files is changed by a FUNCTION call in an argument of a DATASAVE DC or DATALOAD DC statement, this is not detected as a run-time error. The status of the file is checked only at the start of the statement.

References:

DATA LOAD DC OPEN
DATA SAVE DA
DATA SAVE DC OPEN
IF END THEN
Catalog Access - Section 7.3.8 of the Programmer's Guide

DATA LOAD DC OPEN

General Form:

```
     DATALOAD DC OPEN T [file-number,]{file-name        }
                                      {TEMP[,]start, end}
```

Where:

```
     file-name = an alpha-variable or literal-string containing the
                 name of the file to be opened.

     TEMP      = temporary work file being reopened.

     start     = a numeric-expression specifying the starting sector
                 number of the temporary work file.

     end       = a numeric-expression specifying the ending sector
                 number of the temporary work file.
```

NOTE: The use of this statement is not recommended. Refer to Niakwa Data Manager as a better alternative.

Discussion:

The DATALOAD DC OPEN statement is used to open a previously cataloged file for further processing. An error results if the file cannot be located or if it has been scratched. Once open, data may be loaded from or saved to the file using disk catalog operations.

The file name to be opened is indicated by an alpha-variable or literal-string from one to eight characters in length.

Upon execution of the DATALOAD DC OPEN statement, the internal device table entry for the file # specified (file #0, if no file # is specified) is updated with the starting, current, and ending sector numbers of the specified file. Current is initialized to be equal to the starting sector address of the file.

The TEMP parameter is used to specify that a temporary working file be reopened. Temporary files are not cataloged files and must have been created outside the catalog area. The starting and ending sector addresses must be specified. The starting sector address must be greater than the End Catalog sector.

DATA LOAD DC OPEN (cont.)

WARNING--The starting and ending sector numbers of temporary files are stored locally by the CPU. This means that it is possible for the same file space to be allocated simultaneously to more than one user within a multi-user environment. Since the system cannot restrict this from occurring, use special care with this technique.

Examples:

```
0010 DATA LOAD DC OPEN T "FILE1"
0010 DATA LOAD DC OPEN T#1, "DATAFILE"
0010 DATA LOAD DC OPEN T#X, A$
0010 DATA LOAD DC OPEN TEMP 1000,2000
```

Compatibility Issues:

The temporary working file is supported for compatibility purposes but is not recommended.

References:

Catalog Access - Section 7.3.8 of the Programmer's Guide
Extended Diskimages - Section 7.3.10 of the Programmer's Guide

DATA SAVE BA

General Form:

```
DATASAVE BA T[$][file-number,  ] (expr1[,return-value])
                [disk-address, ]
                [<address-var>,]
                                          {alpha-variable}
                                          {literal-string}
```

Where:

```
expr1          = an alpha-variable or numeric-expression.

return-value   = an alpha-variable or numeric-receiver.

alpha-variable = alpha-variable containing data to be saved.
```

NOTE: The use of this statement is not recommended. Refer to Niakwa Data Manager as a better alternative.

Discussion:

The DATASAVE BA statement is used to save the contents of the first 256 bytes of an alpha-variable at a specified sector number.

Presence of the "$" parameter indicates that read-after-write error-checking is to be performed.

Expr1 contains the sector-number to be saved. If expr1 is an alpha-variable, the binary value of the first two bytes is used.

After execution of the statement, the return-value contains the sector-number immediately following the sector number accessed by the operation. If return-value is an alpha-variable, the value is contained in the first two bytes in binary.

DATA SAVE BA (cont.)

Use of an alpha-variable to contain sector addresses results in improper sectors being accessed if extended (greater than 16 MB) diskimages are in use and the sector numbers being accessed are greater than 65355. Refer to Section 7.3.10 of the Programmer's Guide for further programming considerations for use of extended diskimages.

If the alpha-variable to be saved is longer than 256 bytes, only the first 256 bytes are saved. If the alpha-variable is less than 256 bytes, the remainder of the sector is filled with unpredictable values.

DATASAVE BA is a direct access instruction as opposed to a catalog instruction; therefore, the internal device table is not modified by a DATASAVE BA instruction.

Normally, data saved with a DATASAVE BA statement is read back using a DATALOAD BA statement. No control information is saved with the DATASAVE BA statement.

Use of FUNCTIONs as arguments in a DATALOAD or DATASAVE statement, which does not have the platter $OPEN, may result in a disk operation which is not integral. Any implied lock that RTI would issue against the disk is released for the duration of the function call. If the application depends on this implied lock to maintain data integrity, arguments should be evaluated separately from the DATALOAD/DATASAVE statement.

For example:

```
Dim Array1$(22)12,Array2$(22)12

DATA SAVE DA T#1,(X,X)Array2$(),'FunctionResult$,Array2$()
DATA LOAD DA T#1,(X,X)Array1$(),Key$('NextIndex),Array2$()
```

HINT: Function calls should not be used when evaluating arguments in DATALOAD / DATASAVE statements.

DATA SAVE BA (cont.)

Examples:

```
0010 DATA SAVE BA T(Q$,Q$) A$()
0010 DATA SAVE BA T#1, (X-1,X) X$()
0010 DATA SAVE BA T#S, (R$,R) STR(Z$(),257,256)
0010 DATA SAVE BA T/D10, (R$) Z$()
0010 DATA SAVE BA T#4, (X) HEX(43472CFF20)
0010 DATA SAVE BA T/D12, (30) R$()
0010 DATA SAVE BA T<A$>, (30) R$()
```

Compatibility Issues:

The DATASAVE BA statement executes as it does on the Wang 2200, except when the results of the DATASAVE BA would cause data to be written beyond the physical end of the diskimage.

On a Wang 2200, an error I98 (Illegal Sector Address or Platter Not Mounted) would result.

In NPL, the diskimage would be extended to the size needed to accommodate the write operation. If insufficient disk space is available, an error I98 (Illegal Sector Address or Platter Not Mounted) would result.

Any new space allocated would be non-cataloged disk space.

Refer to Section 7.3.4 of the Programmer's Guide for details on dynamic size of disk-image files.

NOTE: **The use of non-cataloged disk space is not recommended. Refer to Chapter 5 of the NPL Supplement for details on support of non-cataloged disk space on the hardware system.**

Use of the address-var parameter is supported only on NPL Revision 3.0 or greater and is not supported on the Wang 2200.

References:

DATA LOAD BA
Dynamic Allocation - Section 7.3.4 of the Programmer's Guide
Direct Access - Section 7.3.9 of the Programmer's Guide

Extended Diskimages - Section 7.3.10 of the Programmer’s Guide
Native OS Files as Diskimage Files - Section 7.3.4 of the Programmer’s Guide

DATA SAVE BM

General Form:

```
DATASAVE BM T[$][file-number,  ] (expr1[,return-value])
                [disk-address, ]
                [<address-var>,]
                                          data-value
```

Where:

```
expr1        = an alpha-variable or numeric-expression.

return-value = an alpha-variable or numeric-receiver.

data-value   = an alpha-variable or literal-string containing the
               data to be saved.
```

NOTE: The use of this statement is not recommended. Refer to Niakwa Data Manager as a better alternative.

Discussion:

The DATASAVE BM statement is used to save the contents of the specified alpha-variable or literal string starting at a specified sector number. If the data to be written exceeds 256 bytes, multiple sectors are written. If the last sector to be written is not filled by the specified alpha-variable or literal, the remainder of the sector is filled with HEX(00). A buffer with zero bytes writes zero sectors.

Presence of the "$" parameter indicates that read-after-write error-checking is to be performed.

Expr1 contains the starting sector-number to be saved. If expr1 is an alpha-variable, the binary value of the first two bytes is used.

NOTE: Use of alpha-variables for the starting sector-number is not advised for diskimage files where the EXT=Y clause has been specified. Refer to Section 7.3.10 of the Programmer’s Guide for further details on extended diskimages.

DATA SAVE BM (cont.)

After execution of the statement, the return-value contains the sector-number immediately following the last sector number accessed by the operation. If return-value is an alpha-variable, the value is contained in the first two bytes in binary.

NOTE: **An error P51 (Variable or Value Too Short) results if an alpha-variable is specified as the return-variable and the sector number exceeds 65535 (extended diskimage in use). Refer to Section 7.3.10 of the Programmer's Guide for further details on extended diskimages.**

DATASAVE BM is a direct access instruction as opposed to a catalog instruction; therefore, the internal device table is not modified by a DATASAVE BM instruction.

Normally, data saved with a DATASAVE BM statement is read back using a DATALOAD BM or DATA LOAD BA statement. No control information is saved with the DATASAVE BM statement.

NOTE: **Use of DATA SAVE BM, as opposed to DATA SAVE BA, could result in significant gains in disk efficiency of the application when:**

1. The amount of data to be written is equal to or an even multiple of the host operating system physical sector size for the disk being written to. This is typically 512 bytes but the actual size may vary. Refer to the appropriate NPL Supplement for details.

2. The starting sector for the operation corresponds to the start of the host operating system sector. Since sector zero of the NPL diskimage is always located at the start of a host operating system physical sector, this can be determined.

For example, with the typical physical sector size of 512 bytes, all NPL even-numbered sectors (0,2,4, etc.) are located at the start of a physical sector.

There are two reasons for the efficiency gain:

1. Since the entire host operating physical sector is to be modified, no pre-read is needed.

DATA SAVE BM (cont.)

NOTE: NPL itself does not perform the pre-read. Rather, this is handled automatically by the operating system.

2. Multiple sectors may be written in one physical operation.

For example, assume that, on an MS-DOS system where the physical sector size is 512 bytes, disk D11 begins at physical sector 1000.

NOTE: The application does not need to know the physical sector numbers–it is given here for purposes of illustration. Further, assume that X$() is dimensioned to a length of 512. The following statements result in physical I/O as indicated:

```
DATA SAVE BM T/D11,(0)X$()
```

Writes one physical sector (sector 1000) of 512 bytes.

```
DATA SAVE BA T/D11,(0)STR(X$(),1,256)
DATA SAVE BA T/D11,(1)STR(X$(),257,256)
```

Will:

a. Reads physical sector 1000

b. Modifies the first 256 bytes of physical sector 1000.

c. Writes physical sector 1000.

d. Reads physical sector 1000.

e. Modifies the second 256 bytes of sector 1000.

f. Writes physical sector 1000.

NOTE: Some of these operations, particularly the second read, would be from buffered data.

DATA SAVE BM (cont.)

Use of FUNCTIONs as arguments in a DATALOAD or DATASAVE statement, which does not have the platter $OPEN, may result in a disk operation which is not integral. Any implied lock that RTI would issue against the disk is released for the duration of the function call. If the application depends on this implied lock to maintain data integrity, arguments should be evaluated separately from the DATALOAD/DATASAVE statement.

For example:

```
Dim Array1$(22)12,Array2$(22)12

DATA SAVE DA T#1,(X,X)Array2$(),'FunctionResult$,Array2$()
DATA LOAD DA T#1,(X,X)Array1$(),Key$('NextIndex),Array2$()
```

HINT: Function calls should not be used when evaluating arguments in DATALOAD / DATASAVE statements.

Examples:

```
0010 DATA SAVE BM T(Q$,Q$) A$()
0010 DATA SAVE BM T#1, (X-1,X) X$()
0010 DATA SAVE BM T#S, (R$,R) STR(Z$(),513,512)
0010 DATA SAVE BM T<X$>, (R$) Z$()
0010 DATA SAVE BM T<A$>, (X) HEX(43472CFF20)
0010 DATA SAVE BM T/D12, (30) R$()
```

Compatibility Issues:

The behavior of DATASAVE BM is different from the Wang 2200 when the results of the DATASAVE BM would cause data to be written beyond the physical end of the diskimage:

On a Wang 2200, an error I98 (Illegal Sector Address or Platter Not Mounted) would result.

In NPL, the diskimage would be extended to the size needed to accommodate the write operation. If insufficient disk space is available, an error I98 (Illegal Sector Address) would result.

Any new space allocated would be non-cataloged disk space.

Refer to Section 7.3.4 of the Programmer's Guide for details on dynamic size of diskimage files.

DATA SAVE BM (cont.)

NOTE: The use of non-cataloged disk space is not recommended. Refer to Chapter 5 of the NPL Supplement for details on support of non-cataloged disk space on the hardware system.

DATA SAVE BM is supported in Revision 3.0 and greater of NPL.

On a Wang 2200, the highest sector number that can be referenced is 65535. In NPL, if the EXT=Y clause is specified on the $DEVICE statement, sector numbers above 65535 can be used. Refer to Section 7.3.10 of the Programmer's Guide for further details on extended diskimages.

Use of the address-var parameter is not supported on the Wang 2200.

References:

DATA LOAD BA, BM
DATA SAVE BA, BM
Dynamic Allocation - Section 7.3.4 of the Programmer's Guide
Direct Access - Section 7.3.9 of the Programmer's Guide
Extended Diskimages - Section 7.3.10 of the Programmer's Guide

DATA SAVE DA

General Form:

```
DATASAVE DA T[$][file-number,  ] (expr1[,return-value])
              [disk-address, ]
              [<address-var>,]
                                       {variable-list}
                                       {END          }
```

Where:

```
expr1         = an alpha-variable or numeric-expression.

return-value  = an alpha-variable or numeric-receiver.

variable-list = {alpha-variable    }[,{alpha-variable    }]...
                {literal-string    }  {literal-string    }
                {numeric-variable  }  {numeric-variable  }
                {numeric-expression}  {numeric-expression}
```

NOTE: The use of this statement is not recommended. Refer to Niakwa Data Manager as a better alternative.

Discussion:

The DATASAVE DA statement is used to save variables, expressions, or literals as logical data records beginning at a specified sector number.

Presence of the "$" parameter indicates that read-after-write error-checking is to be performed.

Expr1 contains the first sector-number to be saved. If expr1 is an alpha-variable, the binary value of the first two bytes is used.

After execution of the statement, the return-value contains the sector-number immediately following the last sector number accessed by the operation. If return-value is an alpha-variable, the value is contained in the first two bytes in binary.

DATA SAVE DA (cont.)

Use of an alpha-variable to contain sector addresses results in improper sectors being accessed if extended (greater than 16 MB) diskimages are in use, and the sector numbers being accessed are greater than 65355. Refer to Section 7.3.10 of the Programmer's Guide for further programming considerations for use of extended diskimages.

Values are saved sequentially as listed. The variable-list may include array designators such as A$() or A(). This indicates that the entire array is to be saved element-by-element. Alpha-values cannot exceed 124 characters.

If the "END" parameter is used, an end-of-file trailer record is written. This record can be used to check for end-of-file with an IF END THEN statement. The "number of sectors used" entry in the catalog is **not** updated, however.

DATASAVE DA is a direct access instruction as opposed to a catalog instruction. That is, the internal device table is not affected by a DATASAVE DA instruction.

Use of FUNCTIONs as arguments in a DATALOAD or DATASAVE statement, which does not have the platter $OPEN, may result in a disk operation which is not integral. Any implied lock that RTI would issue against the disk is released for the duration of the function call. If the application depends on this implied lock to maintain data integrity, arguments should be evaluated separately from the DATALOAD/DATASAVE statement.

For example:

```
Dim Array1$(22)12,Array2$(22)12

DATA SAVE DA T#1,(X,X)Array2$(),’FunctionResult$,array2$()
DATA LOAD DA T#1,(X,X)Array1$(),Key$(’NextIndex),Array2$()
```

HINT: Function calls should not be used when evaluating arguments in DATALOAD / DATASAVE statements.

DATA SAVE DA (cont.)

Examples:

```
0010 DATA SAVE DA T(Q$,Q$) A$, 2*B+1, "TEST", D$()
0010 DATA SAVE DA T#1, (X-1,X) A$(), B$, C(), D()
0010 DATA SAVE DA T$#S, (R$,R) X, Y, Z$()
0010 DATA SAVE DA T/D10, (R$) Z$(), X, A$
0010 DATA SAVE DA T$#4, (X) K$(), STR(A$,2,3), A(2)
0010 DATA SAVE DA T/D12, (30) END
0010 DATA SAVE DA T<A$>, (30) END
```

Compatibility Issues:

The DATASAVE DA statement executes as it does on the Wang 2200, except when the results of the DATASAVE DA would cause data to be written beyond the physical end of the diskimage.

> On a Wang 2200, an error I98 (Illegal Sector Address or Platter Not Mounted) would result.
>
> In NPL, the diskimage would be extended to the size needed to accommodate the write operation. If insufficient disk space is available, an error (Illegal Sector Address or Platter Not Mounted) would result.
>
> Any new space allocated would be non-cataloged disk space.

Refer to Section 7.3.4 of the Programmer’s Guide for details on dynamic size of diskimage files.

Refer to Chapter 5 of the NPL Supplement for details on support of non-cataloged disk space for the hardware system.

Use of the address-var parameter is supported only on NPL Revision 3.0 or greater and is not supported on the Wang 2200.

References:

Dynamic Allocation - Section 7.3.4 of the Programmer’s Guide
Direct Access - Section 7.3.9 of the Programmer’s Guide
Extended Diskimages - Section 7.3.10 of the Programmer’s Guide

DATA SAVE DC

General Form:

```
     DATASAVE DC [$][file-number,]{variable-list}
                                  {END          }
```

Where:

```
     file-number   = a file previously opened with a DATASAVE DC OPEN
                     or DATALOAD DC OPEN statement.

     variable-list = {alpha-variable    }[,{alpha-variable    }]...
                     {literal-string    }  {literal-string    }
                     {numeric-variable  }  {numeric-variable  }
                     {numeric-expression}  {numeric-expression}
```

NOTE: The use of this statement is not recommended. Refer to Niakwa Data Manager as a better alternative.

Discussion:

The DATASAVE DC statement is used to save variables, expressions, or literals as logical data records in a cataloged file. Each execution of the statement saves the next logical data record.

Presence of the "$" parameter indicates that read-after-write error-checking is to be performed.

Values are saved sequentially as listed. The data list may include array designators such as A$() or A(). This indicates that the entire array is to be saved element-by-element. Alpha string variables cannot exceed 124 characters.

If the "END" parameter is used, an end-of-file trailer record is written, the date/time stamp of the file is updated with the current date and time, and the "number of sectors used" value in the trailer record is updated.

DATASAVE DC is a disk catalog instruction, as opposed to a direct access instruction. That is, the starting sector is read from the "current" slot of the internal device table and the "current" slot of the internal device table is updated to the next number following the last sector written by the statement.

DATA SAVE DC (cont.)

Examples:

```
0010 DATA SAVE DC A$, 2*B+1, "TEST", D$()
0010 DATA SAVE DC #1, A$(), B$, C(), D()
0010 DATA SAVE DC $#S, X, Y, Z$()
0010 DATA SAVE DC #X, END
```

Compatibility Issues:

The Date/Time stamp is implemented in Revision 2.00 and greater of NPL.

The Date/Time stamp is not implemented in Wang 2200 Basic-2.

File trailer storage of the Date/Time stamp in catalog data files is implemented in Revision 2.01 of NPL.

File trailer storage of the filename and status of catalog data files is implemented in Revision 3.00 of NPL.

Revision 3.20 of NPL implemented byte 40 of $OPTIONS to allow developers to suppress all trailer section information of catalog data files.

References:

Cataloged Files - Section 7.3.7 of the Programmer’s Guide
Catalog Access - Section 7.3.8 of the Programmer’s Guide
Extended Diskimages - Section 7.3.10 of the Programmer’s Guide

DATA SAVE DC CLOSE

General Form:

```
     DATASAVE DC CLOSE [file-number]
                       [ALL        ]
```

Where:

```
     file-number = a file previously opened with a DATASAVE DC OPEN or
                   DATALOAD DC OPEN statement.
```

NOTE: The use of this statement is not recommended. Refer to Niakwa Data Manager as a better alternative.

Discussion:

The DATASAVE DC CLOSE statement is used to close files previously opened with DATASAVE DC OPEN or DATALOAD DC OPEN.

The DATASAVE DC CLOSE statement initializes the internal device table information to zero but does not internally affect the closed files.

If the "ALL" parameter is specified, all open files are closed.

Examples:

```
0010 DATA SAVE DC CLOSE
0010 DATA SAVE DC CLOSE #1
0010 DATA SAVE DC CLOSE #X
0010 DATA SAVE DC CLOSE ALL
```

Compatibility Issues:

References:

DATA LOAD DC OPEN
DATA SAVE DC OPEN
Catalog Access Methods - Section 7.3.8 of the Programmer’s Guide

DATA SAVE DC OPEN

General Form:

```
    DATASAVE DC OPEN T[$][file-number,]{(scratch-file)new-file}
                                       {(space)new-file       }
                                       {TEMP[,]start,end      }
```

Where:

```
    scratch-file  = the name of an existing scratched program or data
                    file in the specified diskimage.

    space         = a numeric-expression specifying number of sectors
                    required for the new file

    new-file      = the name of the new file being opened

    temp          = the temporary work file being established

    start         = a numeric-expression specifying the starting sec-
                    tor number of the temporary work file.

    end           = a numeric-expression specifying the ending sector
                    number of the temporary work file.
```

NOTE: The use of this statement is not recommended. Refer to Niakwa Data Manager as a better alternative.

Discussion:

The DATASAVE DC OPEN statement is used to create a new cataloged file and open it for further processing. An error results if the file already exists and has not been scratched. Once open, data may be loaded from or saved to the file.

Presence of the "$" parameter indicates that read-after-write error-checking is to be performed.

If an already cataloged filename is specified, the previously allocated space for that file is assigned to the new file. The old file must have been previously scratched. The old filename and the new filename may be the same.

DATA SAVE DC OPEN (cont.)

If the space parameter is specified with a new file name, a new file is created at the current end of the catalog at the sector size specified. If there is insufficient space in the catalog area to create the new file, or if another file of the same name already exists, an error results.

NOTE: The number of sectors actually available for new files is one less than the number specified in the catalog index, since one sector is reserved for system-related information.

The Internal Device Table entry for the file number specified is updated with starting and ending sector locations. Current is set equal to start.

The TEMP parameter specifies creation of a temporary work file, which must be created outside of the catalog area. Starting and ending sector numbers must be specified, with the starting sector number greater than the current catalog END.

WARNING--The starting and ending sector numbers of temporary files are stored locally by the CPU. This means that it is possible for the same file space to be allocated simultaneously to more than one user within a multi-user environment. Since the system cannot prevent this from occurring, use special care with this technique.

The file date/time stamp, filename, file type, and file status fields in the trailer sector are updated by a DATASAVE DC OPEN statement.

Examples:

```
0010 DATA SAVE DC OPEN T$ (200) "FILE1"
0010 DATA SAVE DC OPEN T#1, (100) "DATAFILE"
0010 DATA SAVE DC OPEN T#X, (A) A$
0010 DATA SAVE DC OPEN T$#Q, ("OLDFILE") "NEWFILE"
```

DATA SAVE DC OPEN (cont.)

Compatibility Issues:

The Date/Time stamp is implemented in Revision 2.00 and greater of NPL.

The Date/Time stamp is not implemented in Wang 2200 Basic-2.

The Temporary working file is supported for compatibility purposes but is not recommended.

Storage of file name, type, and status in the trailer sector is supported only in NPL Revision 3.0 or greater and is not supported on the Wang 2200.

References:

Catalog Access Methods - Section 7.3.8 of the Programmer’s Guide
Extended Diskimages - Section 7.3.10 of the Programmer’s Guide
Non-cataloged Disk Space - Chapter 5, NPL Supplement(s)

DATE

General Form:

Form 1:

```
DATE= alpha-expression [PASSWORD {literal-string}]
                                 {alpha-variable}
```

Form 2:

```
alpha-receiver = [$]DATE
```

Discussion:

The DATE system variable is a special system variable which can be used as a receiver (Form 1) to set the system date **or** as a function (Form 2) which allows an alpha-variable to be set to the system's date.

When used as a function (Form 2), the date is returned as an alpha-string, six characters in length. The first two characters are the year, the next two are the month, and the last two are the day of the month.

A System Library function 'CenturyDate$ returns today's date in the format:

```
19YYMMDD
```

with additional logic added to ensure that this value changes to "20YYMMDD" when the value of YY returned by the built-in DATE function is less than 90.

Refer to the System Library Functions Reference for additional information.

Examples:

```
0010 Q$ = DATE
0010 M$() = DATE
0010 DATE = "860620"
0010 DATE = A$
0010 date = "930322" PASSWORD "SYSTEM"
```

Compatibility Issues:

Form 2 of the DATE statement, which READS the date, is fully compatible with Wang 2200 Basic-2 implementation.

DATE (cont.)

The PASSWORD clause is required in Wang 2200 Basic-2. Under NPL, the PASSWORD clause is syntactically supported for compatibility purposes and, if specified, is checked for validity. The system password is "SYSTEM" under NPL and may not be modified.

Operation of this statement may vary on different hardware versions of NPL. Access privileges may be needed to set the system date under certain operating systems. Refer to the appropriate NPL Supplement for details.

References:

DBACKSPACE

General Form:

```
DBACKSPACE [file-number,]{numeric-expression[S]}
                         {BEG                  }
```

Where:

```
numeric-expression = number of records to be backspaced.

S                  = indicates numeric-expression represents num-
                     ber of physical sectors as opposed to logi-
                     cal records.

BEG                = backspace to beginning of file.
```

Discussion:

The DBACKSPACE statement is used with cataloged data files in order to set the "current" value in the internal device for the file number specified to a lower value. It permits backspacing over logical records or physical sectors within the file.

The numeric-expression specifies the number of logical records or physical sectors [S] to be backspaced. The BEG parameter backspaces to the beginning of the file.

When not using the "S" parameter, the number of sectors to subtract from the "current" sector address is determined by actually reading backward through the specified number of logical records.

When using the "S" parameter, the number of sectors specified is subtracted from the current sector address. If this precedes the beginning of the file, the starting sector address is used. Since this operation simply decreases the Current Sector Address in the Internal Device Table by the value of the specified numeric-expression, a disk access is not required.

DBACKSPACE (cont.)

Examples:

```
0010 DBACKSPACE 10
0010 DBACKSPACE #2,BEG
0010 DBACKSPACE X*20
0010 DBACKSPACE #4,2S
0010 DBACKSPACE A*3
```

Compatibility Issues:

References:

Extended Diskimages - Section 7.3.10 of the Programmer’s Guide

DEFFN' Keyboard Input

General Form:

```
DEFFN'integer literal-string [;literal-string]...
```

Discussion:

The keyboard input form of the DEFFN' statement can be used to define commonly used character strings which are entered during INPUT or LINPUT operations. The integer in the DEFFN' statement corresponds to one of the 32 function keys numbered from 0 to 31. Once a character string has been defined, the characters may be recalled by pressing the corresponding special function key.

Character strings must be specified by a literal string inside quotes or by a HEX string, or by a combination of the two. If multiple parts of a string are defined, each part must be separated by a semi-colon (";").

Predefined text strings can be entered any time while using the editor (to enter or edit program text) or during the execution of the INPUT and LINPUT statements. If the character string contains a HEX(0D), the editor responds as if the operator had pressed the RETURN key. Characters after the HEX(0D) are ignored. If quotes are needed in the text string, the HEX representation for quotation marks, HEX(22), can be inserted in the string.

NOTE: Program-defined special function keys can only be used while in DEFFN Mode (not in Edit Mode). Refer to Section 5.4 of the Programmer's Guide for a discussion of Special Function keys in DEFFN Mode and Edit Mode.

WARNING--Use of DEFFN' literals to perform program SAVEs can give unexpected and unpleasant results if the program is currently resolved (HALTed while running), since DEFFN' functions are always referenced in the currently EXECUTING context.

Examples:

```
0010 DEFFN'0"LISTD";HEX(0D)
0010 DEFFN'30"X$=";HEX(22);"UTILITY2";HEX(22);":SCRATCH T X$:
     SAVET()X$";HEX(0D)
```

DEFFN' Keyboard Input (cont.)

These are commonly used subroutines during program development. DEFFN'0 executes a LISTD command. DEFFN '30 is used to SCRATCH and SAVE program in memory to disk. Of course, the program name ("UTILITY2" in the example) has to be specified correctly for the program being worked on.

```
:0010 DIM B$30
:0020 LINPUT "Enter the Directory Path"?-A$
:0030 B$=A$ & "/PLATTER1.BS2"
:0040 $DEVICE(/D20)=B$
:0050 SELECT DISK D20
    : LOAD RUN T"START"
:0100 DEFFN'15 "/BASIC2C/PROGS";HEX(0D)
:RUN
```

In this example, SF'key 15 has been assigned a literal directory path and carriage return. When prompted by the LINPUT statement in line 20, the user needs only to depress SF'key 15 in response.

Compatibility Issues:

Refer to Appendix D of the Programmer's Guide for details on the keyboard equivalences for 'SF keys on the specific terminals.

Wang 2200 Basic-2 requires that DEFFN' be the first statement in a program line (it must immediately follow the line-number). This is NOT a restriction in NPL. The DEFFN' statement may appear in the middle of a multi-statement line.

References:

INPUT
LINPUT
SAVE
SCRATCH
The Line Editor - Section 5.4 of the Programmer's Guide

DEFFN' Subroutine

General Form:

```
     DEFFN'{integer}[(variable [,variable]..)][/PUBLIC][/FORWARD]
            {identifier}
```

NOTE: The use of this statement is not recommended. Refer to FUNCTION or PROCEDURE as a better alternative.

Discussion:

The DEFFN' statement is used to define subroutines which are to be referenced by a specified integer or identifier instead of a line number. Each subroutine is identified by a program statement containing the DEFFN' verb, followed by an integer value from 0 to 65535 or a valid identifier name. The subroutine is ended with a RETURN statement. Subroutines defined with DEFFN' are invoked with the GOSUB' statement.

Unlike normal subroutines, the DEFFN' subroutines support an optional argument list which can be passed values with the GOSUB' statement. This argument list can contain up to 255 numeric-receivers or alpha-variables. The variable types and positions of all variables in the GOSUB' statement must match the types and positions of the DEFFN' statement. When execution of a DEFFN' subroutine is completed (by execution of a RETURN statement), control is transferred to the statement following the GOSUB' statement.

NOTE: Use of more than 16 parameters results in significant performance degradation.

Calling DEFFN's from the Keyboard

DEFFN' subroutines labelled with an integer value of 0 to 31 and 126 to 127 but with no argument list may be invoked from the keyboard during Immediate Mode or during an INPUT or LINPUT operation by pressing the corresponding numbered special function key of a given DEFFN' subroutine. When execution of the subroutine is completed (by execution of the RETURN statement), Immediate Mode is reactivated, the INPUT statement is restarted, or execution proceeds at the statement following the LINPUT statement.

If more than one definition appears for the same function, the definition which appears first in a program listing is used.

DEFFN' Subroutine (cont.)

External DEFFN's:

As of Revision 3.0 of NPL, DEFFN's may be defined in external subroutines developed in other languages. Whenever both an external and an internal DEFFN' of the same number are present, the internal DEFFN' is executed rather than the external DEFFN'. (Refer to Chapter 16 of the Programmer's Guide for further details on external DEFFN's.)

Attributes:

If the keyword PUBLIC is used, the marked subroutine is callable from any module, not just the one in which it's defined. A PUBLIC declaration of the marked subroutine may also appear in the PUBLIC section of the defining module. The FORWARD keyword may be used on PUBLIC declarations to indicate that the body of the function appears later in the module, rather than immediately following the DEFFN' statement. Only one PUBLIC DEFFN' declaration for a given function number (other than FORWARD references) is permitted within the workspace (all modules). A module that declares a PUBLIC DEFFN' may not also declare the DEFFN' as non-PUBLIC.

A DEFFN' [/FORWARD] statement that occurs within a PUBLIC section is implicitly /PUBLIC. In this case, the /PUBLIC keyword is not required but may be entered for clarity.

If both the FORWARD declaration and the definition specify a parameter list, the number and types of all parameters must match, and the variable names in the definition are used (in preference to those in the FORWARD declaration).

Named DEFFN' subroutines and access to these from GOSUB' are supported. Currently, access and duplication rules of named subroutines are identical to those of numbered subroutines. In particular, you may not have DEFFN' inside a FUNCTION body. In addition, if multiple declarations of DEFFN's occur in a program, all but the first are effectively ignored.

DEFFN’ Subroutine (cont.)

NOTE: This may change in a later version to a more strict rule (e.g., duplicate declarations would be flagged as errors unless due to a /FORWARD or /BEGINS).

A DEFFN’ statement is not permitted in the body of a FUNCTION or PROCEDURE.

Examples:

```
0010 DEFFN’250
0010 DEFFN’251(Q,Q1,Q$)
0010 DEFFN’252(STR(A$(),10,200))
0010 DEFFN’1043(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z)
0010 DEFFN’100 /PUBLIC
0010 DEFFN’32768 /PUBLIC/FORWARD
0010 DEFFN’17032(X,X$,Y$)/FORWARD
0010 DEFFN’17032(B,A$(B),STR(R$(B),2))/PUBLIC
0010 DEFFN’MySub(A$,B$)
```

Compatibility Issues:

Use of more than 16 parameters is supported only in NPL Revision 3.0 or greater.

Use of DEFFN’s above ’255 is supported only in NPL Revision 3.0 or greater.

Use of DEFFN’s above ’255 is not supported on the Wang 2200.

In Wang 2200 Basic-2, if a DEFFN’ subroutine is accessed using the keyboard and has parameters, the parameters are requested by a "?" on the screen. In NPL, DEFFN’ subroutines which have parameters may not be accessed from the keyboard. No runtime error is generated. The subroutine simply is not executed.

Wang 2200 Basic-2 requires that a DEFFN’ be the first statement in a program line (it must immediately follow the line-number). This is not a restriction in NPL. The DEFFN’ statement may appear anywhere in a multi-statement program line.

Named DEFFIN’s, PUBLIC, FOWARD, and BEGINS attributes are supported only in NPL Release IV or later.

DEFFN' Subroutine (cont.)

References:

GOSUB
INPUT
LINPUT
RETURN

DEFFN Function Definition

General Form:

```
     DEF FN {letter} (numeric-scalar) = numeric-expression
            {digit }
```

NOTE: The use of this statement is not recommended. Refer to FUNCTION as a better alternative.

Discussion:

The DEFFN statement is used to define a numeric function within a program which can be later called by use of the FN instruction in numeric-expressions.

Each function is identified by a single letter or digit for a total of 36 possible functions within a program. Each function must include a single numeric-expression which is performed when the function is invoked by an FN function call. The numeric-scalar specified is used only as a value in the expression when the function is invoked. The contents of the numeric-scalar are not modified.

The DEFFN statement is not executed if encountered during normal program sequence. The expression in the DEFFN statement is evaluated only when referenced from an FN function call. When the expression is evaluated, the value of the parameter of the FN function is used whenever the numeric-scalar appears in the DEFFN numeric-expression. The value of the numeric-scalar is not changed by the FN function call (the numeric-scalar is a "dummy" variable). Therefore, a DEFFN statement can appear anywhere in a program without affecting normal execution of the program.

If more than one definition appears for the same function, the definition which appears first in the program listing is used.

DEFFN Function Definition (cont.)

Examples:

```
0010 DEFFNA(X)=(X*1.05)+(X-255)
0010 DEFFN9(Y)=Y+2*(Y-9)
0010 DEFFNB(W)=VAL(STR(W$(),W))+T*2
:0010 INPUT Y
:0020 X=FNB(Y)
:0030 PRINT Y,X
:0040 DEF FNB(A)=(A-3)/2
:RUN
? 9
 9                 3
:
```

Compatibility Issues:

References:

Function-name Defined Function

DEFFN@PART

General Form:

```
DEFFN@PART {alpha-variable}
           {literal-string}
```

NOTE: This statement is supported for Wang compatibility reasons only and its use in new development is not recommended.

Discussion:

The syntax of this statement is supported only for compatibility with Wang 2200 Basic-2. No operation is performed when this statement is encountered at execution time.

Replacing Global Partitions

The use of global partitions to support shared program code and variables is not supported by NPL. Where the primary purpose of the global partitions is simply to share code, it may be possible to place the global partition's code into an NPL library module instead. Some program changes are required, such as adding an appropriately designed PUBLIC section to define the publicly accessible DEFFN' entry points and global variables, and replacing SELECT@PART statements in programs that use the global partition with an INCLUDE statement to make the library module available. Also, any variable initialization which is done before the DEFFN@PART statement may be placed in a /MAIN procedure to ensure it is executed before library functions are called.

Where the primary purpose of global partitions is to share variables (e.g., for record locking), the application must be rewritten to be compatible with networked environments, where the sharing of information is done only through commonly accessed files.

In the Wang 2200 MVP/CS environment, application software (e.g., KFAM) that uses global partitions for sharing variables usually also supports some alternative method which uses disk files. These alternative methods are often provided to allow for use with larger systems that involve multiple CPU's and multiplexed disk drives, since, in these environments, the use of "system-wide" shared variables is not possible. Configuring the application to use this alternative disk-based method may allow NPL applications to run without involving substantial changes.

DEFFN@PART (cont.)

Examples:

Compatibility Issues:

This instruction, when executed on a Wang 2200 MVP, signals the specified partition as a "global" partition. Global partitions are not supported under NPL.

On a Wang 2200, variable names which are previously undeclared and appear after a DEFFN@PART statement in a program are assumed to be variables of another partition, are not allocated space in the partition, and need not be declared (if arrays) in a DIM or COM statement. All array-variables must be declared under NPL and all variable space must be allocated. DEFFN@PART has no effect on program resolution of variables.

On a Wang 2200, an optional FOR terminal# [,terminal#] clause is permitted to restrict access to a global partition. This syntax is not supported by NPL.

References:

SELECT @PART
GOSUB’

DELETE

General Form:

```
     DELETE T [file-number,     ] file-name [,file-name]...
              [disk-address,    ]
              [<alpha-variable>,]
```

Where:

```
     file-name = an alpha-variable or literal-string containing the
                 name of the existing cataloged file to be scratched.
```

Discussion:

The DELETE statement is used to remove the index entry of a file or files. The purpose of this statement is to provide a convenient method of eliminating file names which are no longer used.

All file-names specified must be present and scratched. DELETE does not remove a non-scratched file-name.

The actual contents of the file are not affected by DELETE. However, once a file has been removed, the index entry for the file cannot be reconstructed by any NPL statement. Therefore, the contents of the file are effectively lost.

In most cases, the space that was used by the files deleted does not become available for reuse until a MOVE operation or equivalent utility is executed. However, when the file deleted is the last file on the diskimage (end sector of the file is equal to Current End for the diskimage), Current End is set to the start sector of the file minus one.

NOTE: Where multiple files are to be deleted, they are processed one by one in the order specified. For example, where the first file specified is the second from last file on the diskimage and the second file specified is the last file on the diskimage, Current End is decremented only for the second file specified. When the first file is processed, it is not the last file on the diskimage.

DELETE (cont.)

Examples:

```
0010 DELETE T"START",Q$
0010 DELETE T X$,X1$,X2$
0010 DELETE T#Y,"SP MENU"
:DELETE T"START"
:DELETE T/D32,"START","SP START","SECURITY"
:DELETE T<A$>,"PROGRAM"
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

DELETE is not supported on the Wang 2200.

References:

SAVE
DATA SAVE DC OPEN
SCRATCH
UNSCRATCH

$DEMO

General Form:

Form 1:

```
$DEMO = alpha-expression
```

Form 2:

```
alpha-receiver = $DEMO
```

Discussion:

$DEMO is a system variable which allows for the generation of "self demonstrating" software. With this technique, live software is operated by using keystrokes and informational text from an ASCII text file, as opposed to the keyboard.

Form 1:

When a value is assigned to $DEMO, the value must be a valid native operating system file specification or blank spaces. If not blank, all keyboard input is redirected from the specified file. No keyboard keys function, with the following exceptions:

- SPACE BAR - this is used to continue with the Demo script when it has been halted by a special screen display.
- CANCEL - pressing CANCEL when a BOX is displayed and the non-interpretive RTP is running exits the Runtime. This is an intended result, designed to allow a user who is unfamiliar with the software's structure to exit from a software demonstration at any time. The interpretive RTI returns the keyboard to normal use, to allow testing and correction of $DEMO files.
- ARROW Keys - may be used to interactively reposition special screen displays at execution time.
- PLUS and MINUS Keys - May be used to speed up or slow down the rate of keystroke entry from the demo script file.

$DEMO (cont.)

When the end of the demo script file is reached, the value of the $DEMO variable is not removed, but keystrokes and text are no longer read from the file. If the demo script file specified by the $DEMO system variable does not exist, or is invalid, or is blank, the keyboard input returns to normal.

As of Revision 4.0 of NPL, $OPTIONS byte 42 may be set to aid in the debugging of sequencing problems encountered by a $DEMO script. Refer to $OPTIONS for details.

Form 2:

The current status of the $DEMO system variable may be examined by using Form 2 of the $DEMO statement. In addition, byte 19 of $MACHINE may be examined to determine whether or not the next keystroke is generated from a $DEMO script.

As of Revision 3.0 of NPL, keyboard logging may be used to generate files suitable for use with $DEMO. Refer to $DEMO, Chapter 12 of the Programmer's Guide. for details on the structure of the demo script file. Refer to SELECT LOG for further details on keyboard logging.

Examples:

```
    0010 $DEMO="/BASIC2C/SCRIPT1.DAT"
```

In this example, the /BASIC2C/SCRIPT1.DAT file would be redirected for keyboard input as the Demo script file.

Compatibility Issues:

This statement is supported only with Release 2.0 or greater.

This statement is not valid in Wang 2200 Basic-2.

Use of $MACHINE to determine the status of $DEMO redirection and generation of demo script files by keyboard logging are both supported only on NPL Revision 3.0 or greater.

References:

$MACHINE
$OPTIONS
SELECT LOG
$DEMO **-** Chapter 12 of the Programmer’s Guide

$DET

General Form:

```
alpha-receiver = $DET(numeric-expression)
```

Discussion:

The $DET function allows program inspection of the device addresses currently defined in the Device Equivalences Table (DET). The numeric-expression is used to specify which entry in the DET is accessed. The number of DET entries may range from 16 to 255, as established by the /D RunTime option. Attempting to access a DET entry greater than 65535 results in a P34 (Illegal Value) error.

The value returned by $DET is a three-byte, alpha-numeric value representing the device-address defined by a prior $DEVICE Statement. If a given slot in the DET is currently not used, spaces are returned.

NOTE: The device-class byte of the device-address returned by $DET is subject to the same translation as shown by LISTDT. For instance, print class addresses yzz are always returned as 0zz, disk addresses 3x0 are always returned as Dx0, and disk addresses Bx0 are always returned as Dx1.

The actual host operating system device equivalence for the device-address returned may then be accessed by use of the $DEVICE statement.

For example, the following program displays all devices currently defined in the DET:

```
0010 DIM A$3,B$50,M$64
0020 M$=$MACHINE
0030 M=VAL(STR(M$,16,1)
    :REM Number of DET entries is in $MACHINE
0040 PRINT "DEVICE ADDRESS";TAB(20);"DEVICE EQUIVALENCE"
0050 FOR X=1 TO M
0060    A$=$DET(X)
0070   IF A$<>" "
0080         B$=$DEVICE(A$)
0090         PRINT A$;TAB(20);B$
0100   END IF
0110 NEXT X
```

$DET (cont.)

NOTE: The order of entries returned by $DET() has no special significance and may be changed when new values are assigned to DET entries. If there are n valid $DEVICE values currently defined, all valid addresses correspond to $DET(1) through $DET(n).

Examples:

```
0010 A$=$DET(1)
0010 STR(A$,4,3)=$DET(X)
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

$DET is not supported on the Wang 2200.

References:

$DEVICE
Device Equivalence Table - Section 7.2.2 of the Programmer's Guide

$DEVICE

General Form:

Form 1:

```
$DEVICE({device-address})=alpha-expression
        {file-number    }
        {alpha-variable }
```

Form 2:

```
alpha-receiver=$DEVICE({device-address})
                       {file-number   }
                       {alpha-variable}
```

Where:

alpha-variable	=	contains a valid NPL device-address or file-number (with no preceding slash).
alpha-expression	=	an alpha-expression which evaluates to a native operating system file-specification or device-specification followed by one or more optional clauses. The total resultant value of the alpha-expression should not exceed 50 characters in length. One blank space separates the file or device-name and each optional clause.

```
clause           = [1.2={N,Y}           ]
                   [1.4={N,Y}           ]
                   [2.8={N,Y]           ]
                   [360={N,Y}           ]
                   [720={N,Y}           ]
                   [ALF={N,Y}           ]
                   [ERR={N,Y}           ]
                   [EXT={N,Y}           ]
                   [LCL={N,Y}           ]
                   [PES=numeric-constant]
                   [SES=numeric-constant]
                   [TMO={N,Y}           ]
                   [XLA={N,Y}           ]
```

$DEVICE (cont.)

Discussion:

The $DEVICE assignment statement provides a method of changing or inspecting a specific entry in the Device Equivalence Table. The Device Equivalence Table establishes an equivalence between NPL device addresses and the native operating system files or devices. The number of Device Equivalence Table entries is set to 16 by default but may be set higher (up to a maximum of 255) by use of the /D startup option. Refer to Section 7.2.2 of the Programmer's Guide for details on the Device Equivalence Table.

Form 1: Changing the Device Equivalence Table

Form 1 can be used to modify the device equivalence table of the $DEVICE statement.

If the result of the alpha-expression used in the assignment is blank, the specified device-address is removed from the table. Otherwise, it replaces the current native name that the NPL device address is currently mapped to, and all subsequent I/O is performed to the new device. The alpha-expression should equate to a valid native operating system file or device specification. No checking is done for validity of the specification until an attempt is made to access the device, at which time a P48, Invalid Device Specification, occurs if the device is not available.

For example:

```
0010 $DEVICE(/D11)="/other/platter1.bs2"
   : $DEVICE(/D12)="/data/platter1.bs2"
   : SELECT DISK /D11
   : LOAD RUN
```

This changes the device equivalents so that subsequent references to the disk addresses /D11 and /D12 refer to the diskimage files "/other/platter1.bs2" and "/data/platter1.bs2" respectively. It subsequently loads and runs the "START" program from the "/other/platter1.bs2" diskimage.

Execution of a $DEVICE statement (Form 1) which specifies a file or device which has already been opened by prior access causes that file or device to be closed. Whether the file was previously open or not, the first actual I/O statement following execution of the $DEVICE statement which is directed to the NPL address causes the file to be opened.

$DEVICE (cont.)

Form 2: Inspecting the Device Equivalence Table

The current state of the device equivalence table may be inspected by using Form 2 of the $DEVICE statement.

An alpha-receiver can be set equal to the current entry in the Device Equivalence Table for the specified device-address or file-number. This can be a useful programming tool when inspecting and retaining the current device equivalence table for later restoration.

As of Revision 3.0 of NPL, additional statements may be used to examine the status of the Device Equivalence Table:

- $MACHINE contains information about the maximum size of the DET and the number of entries currently in use.
- $DET may be used to examine the DET in physical order to determine which device-addresses are assigned.

Refer to $MACHINE and $DET for details.

For example, the following is executed immediately after startup (using default device equivalences established by the RunTime Program):

```
:0010 DIM A$50,B$50
    : A$=$DEVICE(/D11)
    : B$=$DEVICE(/215)
    : PRINT "/D11=";A$
    : PRINT "/215=";B$
:RUN
/D11=platter1.bs2
/215=/DEV/PRN
```

NOTE: The start-up device equivalences are dependent on the native operating system. Refer to the appropriate NPL Supplement for details.

If the device-address specified is not currently defined in the device equivalence table, the alpha-receiver is set equal to blank spaces.

$DEVICE (cont.)

Optional Clauses:

One or more optional clauses may be specified in the $DEVICE statement. These optional clauses are used to set specific I/O options (each option is discussed below) for accessing the native operating system file or device.

NOTE: **Most options are specific to only one device class (print or disk). Use of device class specific options with the incorrect device class is typically ignored. Also, some options are mutually exclusive. Use of mutually exclusive options in the same $DEVICE statement may result in ambiguous results.**

The effect of many options is highly operating system-dependent. In addition, it is possible that new options are defined for new ports of NPL. Please refer to the NPL Supplements for complete details on the actual options available and their effect for the operating system.

"Raw" Diskette Clauses:

The five "raw" diskette clauses, 1.2=Y, 1.4=Y, 2.8=Y, 360=Y, and 720=Y, instruct the RunTime program to treat the associated native operating system device name as a "raw" device with a particular format:

1.2=Y	5-1/4" high density diskette (1.2 MB capacity)
1.4=Y	3-1/2" high density diskette (1.44 MB capacity)
2.8=Y	3-1/2" diskette (2.88 MB capacity)
360=Y	5-1/4" diskette (360K capacity)
720=Y	3-1/2" diskette (720K capacity)

For example, refer to the following statement under MS-DOS:

```
$DEVICE(/D20)="A: 1.2=Y"
```

This instructs the RunTime program to treat drive A as a "raw" 1.2 MB diskette.

Support of "raw" diskette access is extremely hardware and operating system-dependent. On some operating systems, various "raw" diskette formats are accessed by use of special device names rather than the $DEVICE clauses described above. Refer to Section 5.2 of the NPL Supplement for details on "raw" diskette support and naming conventions on the operating system.

$DEVICE (cont.)

The ALF Clause

The ALF option when specified as part of a $DEVICE statement controls output options for print type devices. The form of the ALF option may be set to "Y" or "N". "Y" indicates that the special output options are to be used. "N" indicates that the special output options are not to be used. The ALF option is added on to the end of the alpha-expression.

For example:

```
0010 $DEVICE(/215)="/dev/prn ALF=N"
```

Special output options generally control end of line sequences for printers.

For example, under the MS-DOS versions of NPL, a linefeed (HEX(0A)) is automatically generated by the RunTime whenever a carriage return (HEX(0D)) is encountered in print output. Setting ALF=N in a $DEVICE statement suppresses the generation of automatic line-feeds under MS-DOS.

NOTE: **The operator cannot override the ALF specification by using the Auto Linefeed function of the PRINT CONTROL screen in the HELP processor. However, execution of a subsequent $DEVICE statement with the ALF option resets ALF to the value specified.**

If ALF is not specified, the initial default value of "Y" is used.

The ALF=N clause is commonly used when sending binary sequences (graphics, for example) which may contain HEX(0D)'s (which are not end-of-line markers), or when performing overstrike printing on the same line.

Refer to Section 5.5 of the appropriate NPL Supplement for hardware or operating system-specific details regarding $DEVICE and the ALF printer option.

The ERR Clause:

The $DEVICE clause, ERR=[Y,N], controls whether or not NPL errors are generated if the host operating system reports an error when writing print class output to a disk file.

$DEVICE (cont.)

The default value of "N" means that a NPL error is never generated during a print class output operation. A value of "Y" means that NPL generates disk class errors for print class output when the print address is defined using $DEVICE as a disk file and the host operating system reports an error. All possible disk type errors are reported, but error code D81 (File Full) is the only error code used. Such errors are recoverable and may be trapped by an ERROR statement.

NOTE: The ERR clause is meaningful only for print class devices, assigned to direct output to a disk file.

For example:

```
0010 $DEVICE(/215)="TEXT.DAT ERR=Y"   :REM Enable error detection
0020 SELECT PRINT 215
0030 PRINT "THIS IS A TEST"<
    :ERROR E=ERR
```

If an error occurs during a $GIO statement, any character count or LRC registers in the arg-2 variable may contain invalid values.

The EXT Clause:

The EXT clause is meaningful only for disk class devices. EXT=Y indicates that the diskimage file is to be treated as an extended diskimage file. The default value of EXT=N indicates that the diskimage is to be treated as a non-extended diskimage. An extended diskimage is a diskimage file that may exceed 16 MB. In accessing extended diskimage files, RTP uses bytes 7&8 of each index entry as the high-order byte of the sector address. In accessing non-extended diskimage files (EXT=N), bytes 7&8 of index entries are ignored and all sector addresses are treated as two byte addresses. Use of extended diskimages has serious implications for the application program. These implications are discussed in greater detail in Section 7.3.10 of the Programmer's Guide.

For example:

```
$DEVICE(/D11)="PLATTER1.BS2 EXT=Y"
```

defines D11 as an extended diskimage file named PLATTER1.BS2.

$DEVICE (cont.)

The LCL Clause:

The LCL clause is valid only for print class device. "Y" is used to specify that a local printer attached to the terminal in use is to be used for print output directed to the device address specified. The default value of "N" indicates that the printer is not a local printer. Local printers are supported on most serial terminals which have local printer capability. Refer to Appendix D of the Programmer's Guide for details on local printer support for the terminals being used.

When a local printer is accessed by use of the LCL clause, operation of the local printer is transparent to the native operating system. The RunTime Program automatically sends required control codes to the terminal to switch between screen and printer output as required by the application.

NOTE: **An $OPEN directed to device address defined as a local printer does not "hog" that address on a system wide basis (as it would for non-local printers). Thus, multiple terminals may use the same device address for local printers. Any valid print class device address may be defined as a local printer.**

For example:

```
$DEVICE(/204)=">1 LCL=Y"
```

This defines address 204 as a local printer on the terminal in use under SuperDOS. Refer to Section 5.5 of the NPL Supplements for further details on the device-name to use for the operating system.

The PES and SES clauses:

The PES= and SES= clauses are used to specify the primary and secondary extent sizes, in units of 256 byte blocks, to be used when a file is created. The PES= and SES= clauses are valid for both disk class devices and print class devices. This clause is meaningful only on operating systems where disk space is allocated in fixed size extents and the size of primary and secondary extents can be set under program control. On these operating systems, the numeric-value specified by the PES and SES clauses overrides built in logic for determining the extent size to use. On other operating systems, this clause is syntactically supported but has no effect. Refer to Section 5.3 of the appropriate NPL Supplements for information about fixed disk allocation for the operating system and the default method used to determine extent sizes where applicable.

$DEVICE (cont.)

NOTE: The extent size for a file can only be set at the time of file creation. For disk class devices, files are created by SCRATCH DISK and MOVE. Also, for disk class devices, if the file already exists, existing extent sizes are used and any PES or SES clause specified is not used. For print class devices, files are created when output is directed to a disk file that does not previously exist.

In the event that the specified primary or secondary extent size cannot be allocated, the RunTime automatically attempts to use built in defaults before aborting the operation. If built in defaults also cannot be allocated, a NPL error results.

For example, refer to the following under SuperDOS:

```
$DEVICE(/D11)="PLATTER1.BS2 PES=10000 SES=1000"
SCRATCH DISK T/D11,LS=10,END=12000
```

This specifies that the primary extent size for PLATTER1.BS2 equals 10000 256 byte blocks (or 2,560,000 bytes) and that the secondary extent size equals 1000 256 byte blocks (or 256,000 bytes).

The TMO Clause:

The TMO clause is valid only for print class devices. This clause controls whether or not a multi-character input operation ($GIO microcommand C620) returns with zero bytes read if there are no bytes present at the specified device. This is useful for communications applications which use the C620 microcommand to read data from a serial port. A value of "Y" indicates that time-out is to occur and that the C620 microcommand returns with zero bytes if no bytes are present at the specified port. The default value of "N" indicates that no time-out is to occur and that the C620 microcommand waits for bytes to be present, thus "hanging" the application if no bytes are present.

Limited serial communications support is operating system-dependent and may not be suitable for all communications requirements. On some operating systems, enhanced asynchronous communications support is available with use of the Niakwa Science and Communications Drivers (SCD) Package. Refer to Section 5.7 of the NPL Supplements for details on limited serial communications support and the availability of the SCD Package on the operating system. In addition, refer to Section 7.8 of the Programmer’s Guide for further information on limited serial communications techniques.

$DEVICE (cont.)

For example:

```
$DEVICE(/219)="com1 TMO=Y"
```

This defines address 219 as the com1 port on an IBM PC and specify that time-outs are to occur on input operations from this device.

The XLA Clause:

The XLA clause is valid only for print class devices. A value of "Y" indicates that printer translation is to take place for output directed to the specified device address. The default value of "N" indicates that no printer translation is to take place. Refer to Section 7.7.7 of the Programmer's Guide for further details on printer translation.

For example:

```
$DEVICE(/215)="/dev/prn XLA=Y"
```

This indicates that printer translation is to take place for output directed to address 215.

Examples:

```
0010 $DEVICE(/D32)="/niakwa/progs/platter1.bs2"
0010 $DEVICE( A$)="/basic2c/data/platter1.bs2"
0010 $DEVICE(X$)=Y$
0010 $DEVICE(#1)="/usr/BASIC2C/platter1.bs2 EXT=Y"
0010 $DEVICE(/217)="/basic2c/spool.dat SES=500 ERR=Y XLA=Y"
0010 A$=$DEVICE(B$)
0010 A$=$DEVICE(/D32)
0010 A$=$DEVICE(/217)
0010 FOR I=1 TO 5
    :   J$(I)=$DEVICE(K$(I))
    : NEXT I

0010 $DEVICE(/D11)="/progs/platter1.bs2 EXT=Y"
   : $DEVICE(/D12)="/data/platter1.bs2"
   : $DEVICE(/215)="/dev/prn ALF=N"
   : $DEVICE(/216)="spool.dat ERR=Y"
   : SELECT DISK /D11
   : LOAD RUN
```

DEVICE (cont.)

This example specifies disk address D11 as extended diskimage /progs/platter1.bs2, disk address D12 as non-extended diskimage /data/platter1.bs2, printer address 215 as /dev/prn (with optional ALF clause equal to N), and printer address 216 as a native operating system file named spool.dat with error detection enabled. Information sent to printer address 216 is redirected to this data file.

Compatibility Issues:

$DEVICE is not supported in Wang 2200 Basic-2.

Use of optional clauses is supported on NPL revisions as follows:

1.2	2.01 or greater
1.4	3.00 or greater
2.8	4.00 or greater
360	2.01 or greater
720	3.00 or greater
ALF	2.00 or greater
ERR	3.00 or greater
EXT	2.01 or greater
LCL	2.01 or greater
PES/SES	3.00 or greater
TMO	2.01 or greater
XLA	2.01 or greater

Use and functionality of many optional clauses is operating system-dependent. Refer to Chapter 5 of the NPL Supplements for details.

References:

$DET
$MACHINE
The Device Equivalence Table - Section 7.2.2 of the Programmer's Guide
The /D option - Section 2.4.2 of the Programmer's Guide
"Raw" diskette handling - Section 5.2 of the NPL Supplements
Printer Devices - Section 5.5 of the NPL Supplements
Serial Port - Section 5.7 of the NPL Supplements

DIM

General Form:

```
     DIM dim-element[,dim-element]...
```

Where:

```
     dim-element = [-]
                   {numeric-id       [=initial-num-value]      }
                   {numeric-array-id (sub1[,sub2])             }
                   {alpha-id$[length][=initial-str-value]      }
                   {alpha-array-id$(sub1[,sub2])[length]       }

     sub1, sub2   = numeric-expression which evaluates to a value in
                    the range from 0 to 65535.

     length       = numeric-expression which evaluates to a value in
                    the range from 1 to 65535.  If length not
                    specified, default to 16.
```

Discussion:

The DIM statement is used to declare variables within NPL programs, and possibly also to assign an initial value to the variable.

If the DIM statement does not occur within a PUBLIC section or within a FUNCTION or PROCEDURE body, the variables are declared as module private non-common variables.

If the DIM statement occurs within a PUBLIC section, the variables are declared as PUBLIC variables (see DIM/PUBLIC).

If the DIM statement occurs within a FUNCTION or PROCEDURE body, the variables are declared as RECURSIVE variables (see DIM/RECURSIVE).

Non-common variables are cleared by execution of:

- The CLEAR command,
- The CLEAR V command,

DIM (cont.)

- The CLEAR N command,
- The LOAD RUN command,
- A program overlay,
- The RUN command (or statement).

Either dimension of an array may be specified as containing up to 65535 elements and the length of any variable may be specified up to 65535 bytes. However, the maximum total number of array elements must not exceed 65535 and the total size of the array (number of elements * length) must not exceed 65535 bytes.

On 32-bit hardware platforms, NPL supports access to larger arrays, and on these dimensions of arrays and string lengths may be permitted to be larger than 65535 (up to 4294967295, or the value imposed by the operating system due to real or virtual memory limits).

If a numeric-expression is used in dimensioning variables, any variables contained in the numeric-expression must be declared prior to being used in the DIM statement. Useful numeric-expressions normally used as terms only constants, common variables, previously declared scalar variables which have been assigned an initial value, function values for FUNCTIONs in previously INCLUDEd modules and built-in functions such as SPACE in some combination.

For syntactical reasons, if an expression is used for the length of an alpha-scalar, it may not begin with "(".

Dimensioning of arrays to zero elements is supported. This is a useful programming tool for dynamically establishing array size based on available memory.

For example:

```
0010  DIM Table$(2)20,OK$32,SizeWas
0020  FUNCTION 'Try To Expand(/POINTER RefToArray$,Newsize
    : DIM Oldsize
    : Oldsize=LEN(STR(Table$()))/LEN(STR(Table$()))
    : MAT REDIM Table$(Newsize)20
    : RefToArray$=ALL("X")        :;
```

DIM (cont.)

Initial Values for Scalar Variables

DIM statements are extended to permit initial value assignment to scalars when the variable is created. If no initial value is defined, the default value is 0 for numerics, blank for strings.

NOTE: Initial values are only assigned when a variable is created. If the variable already exists (i.e., is COMmon, or the program was RUN with line numbers to avoid clearing non-common variables), no initial value is assigned.

The allocation of variables in a DIM statement is implicitly /STATIC if the DIM statement is not within a function body or PUBLIC section. The allocation of variables in a DIM statement is implicitly /RECURSIVE if the DIM statement is within a function body. The allocation of variables in a DIM statement is implicitly /PUBLIC if the DIM statement is within a PUBLIC section. Refer to DIM /STATIC, DIM /RECURSIVE and DIM /PUBLIC for details and restrictions of each allocation type.

As with all expressions which are evaluated at resolve time, the initial values may only reference previously declared values, or FUNCTIONs which are declared in other, previously INCLUDEd modules.

Default variable declaration is assumed by NPL when a variable reference appears without any previous explicit declaration. Here, NPL takes one of two actions:

a. If $OPTION byte 38 is set to HEX(01), an error occurs. All variable references must be preceded by a declaration.

b. If $OPTION byte 38 is set to HEX(00), a variable may be declared by default in some cases, depending on the context in which the first variable reference in the program appears, according to the following table:

Location of First Variable Reference	**Default Allocation Type**
Within a function body	Not legal; error occurs
Outside all function bodies	DIM/STATIC

DIM (cont.)

NOTE: Constant variables must always be explicitly declared.

Examples:

```
0010 DIM Ratio,E=EXP(1)
0010 DIM FileName$8,Default_City$30="Moose Jaw"
0010 DIM CornYields(_NUMBER_OF_MONTHS_KEPT_ON_RECORD)
0010 DIM Buffer$(_MAX_OPEN_FILES)512

0010 DIM A$(SPACE-20000)1,B$((SPACE-1000)/C2)C2,C$(MAX(256,J))1
```

This declares variables which use, respectively, all but the last 20K of memory, the remaining memory (after A$()) less 1K set up as elements of C2 bytes each (assuming C2 is a common variable) and an array with either J bytes or 256 bytes, whichever is larger (again, J is assumed to be a common variable).

```
0010 DIM X$24, Q$(4)4, X(4,4)
```

This will define one variable X$ to 24 bytes, and two arrays: Q$() - four elements, four bytes each and X() - a numeric two dimensional array four by four.

Compatibility Issues:

The Wang 2200 Basic-2 limitation of 124 characters on the length of a scalar has been extended to 65535 (larger on 32-bit platforms). (Be aware, however, that 124 bytes is still the largest scalar variable length which can be saved using a DATASAVE DC statement).

The memory overhead for variables is greater under NPL than in a Wang 2200 Basic-2:

Variable Type	**Overhead on Wang 2200**	**Overhead under NPL**	
		One dimension	**Two dimensions**
Numeric Scalar	4	8 (10)	N/A
Numeric Array	6	12 (16)	14 (18)
Alpha Scalar	5	10 (12)	N/A
Alpha Array	7	14 (18)	16 (20)

DIM (cont.)

NOTE: Numbers in parentheses () are for 32-bit platforms. NPL always allocates variable memory in units of 16 bytes. Consequently, in addition to the indicated overhead and the defined variable size, up to 15 bytes of overhead may be required to allocate the variable. The above figures are for statically allocated (not /RECURSIVE) variables.

Variables which are designated by long identifiers require some additional overhead, which is deducted at program load time.

Wang 2200 Basic-2 does not allow expressions to be used as dimension sizes. Wang 2200 Basic-2 allows only constant and numeric-scalar common variables to be used as variable dimensions.

Wang 2200 Basic-2 does not allow array-variables to be dimensioned with 0 elements.

Constant variables and initial values are only supported by NPL Release 4.0 or later.

References:

COM
DIM /PUBLIC
DIM /RECURSIVE
DIM /STATIC
FUNCTION
$OPTIONS
PROCEDURE
PUBLIC

DIM Constant Variable Declarations

General Form:

```
     DIM  [/PUBLIC] const-variable[,const-variable]...
          [/STATIC]
```

Where:

```
    const-variable = {_numeric-id        =initial-num-value }
                     {_alpha-id$[length] =initial-str-value }

    length          = numeric-expression which evaluates to a value
                      in the range from 1 to 65535.  If length not
                      specified, default to 16.
```

Discussion:

DIM constant declarations allow for declarations of scalar and numeric variables which are evaluated at resolve time. Identifiers which are constant class are always preceded by an underscore ("_").

NOTE: As with all expressions which are evaluated at resolve time, the initial values may only reference previously declared values, or FUNCTIONs which are declared in other, previously INCLUDEd modules.

The compiler attempts to detect syntax which could modify the value of variables declared as constants, and flags such syntax as an error.

There may be some syntax which could result in modifying a constant that cannot be detected at compile time (e.g., passing a constant variable as argument to FUNCTION where /POINTER parameter is required) and these are flagged at resolve time.

There may be some syntax which could result in modifying a constant that cannot be detected at resolve time (e.g., passing a constant variable as argument to indirectly named FUNCTION where /POINTER parameter is required) and these are flagged at execution time.

NOTE: A constant identifier which occurs within a PUBLIC section is PUBLIC by default.

DIM Constant Variable Declarations (cont.)

CONSTANTs may also be used as part of numeric expressions used to specify the number of array elements and the length (for alphanumerics).

For example:

```
10 DIM _NumberOfElements=100
20 DIM _ElementLength=4
30 DIM Array$(_NumberOfElements)_ElementLength*2
 : RUN
 : LIST DIM Array$(
DIM Array$(100)8
```

NOTE: Constants can also be used with the COM statement under Release IV.

In addition, a DIM statement can contain a mix of constant and non-constant variables, for example:

```
0010 DIM X, _Y, Z, _Apples
   : DIM /PUBLIC Buffer$512, _SystemId$ = "Windows"
```

Examples:

```
0010 DIM _CGA_RED=4,_CGA_GREEN=2,_CGA_BLUE=1
0010 DIM _BRIGHT$5=HEX(020400020E),_NORMAL$1=HEX(0F)
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

DIM
COM

DIM /PUBLIC

General Form:

```
     DIM /PUBLIC dim-element[,dim-element]...
```

Where:

```
     dim-element = {numeric-id       [=initial-num-value]      }
                   {numeric-array-id (sub1[,sub2])             }
                   {alpha-id$[length][=initial-str-value]      }
                   {alpha-array-id$(sub1[,sub2])[length]       }

     sub1, sub2   = numeric-expression which evaluates to a value in
                    the range from 0 to 65535.

     length       = numeric-expression which evaluates to a value in
                    the range from 1 to 65535.  If length not
                    specified, default to 16.
```

Discussion:

This statement declares a list of variables that are added to the workspace PUBLIC variable space (if they do not already exist).

Variables in the PUBLIC variable space may be referenced by name in all modules. A PUBLIC variable may be referenced by name in a module provided:

- The variable appears in a DIM PUBLIC statement located in the module or in a PUBLIC section which the module USES.

All declarations of string and array variables in the PUBLIC variable space must agree as to length and number of elements.

- When the reference occurs within the body of a function or procedure, no RECURSIVE or STATIC variable or parameter of the same name has been declared in the current function.

- When the reference does not occur within the body of a function or procedure, no STATIC variable of the same name has been declared in the module.

DIM / PUBLIC (cont.)

The declaration of the variable must always appear before the reference to the variable.

DIM statements which occur within a PUBLIC section are implicitly DIM /PUBLIC. In this case, the /PUBLIC keyword is optional but may be entered for clarity.

NOTE: Values assigned to /PUBLIC variables survive only as long as the defining module remains resolved.

Examples:

```
0010 DIM /PUBLIC Inited,FishName$16,B,A(VectorSize)
0010 DIM /PUBLIC NameList$(20)45
    :DIM /PUBLIC Temporary,Access$256
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

LIN's are supported in Release IV or greater.

References:

DIM /RECURSIVE

General Form:

```
DIM /RECURSIVE dim-element[,dim-element]...
```

Where:

```
dim-element = {numeric-id       [=initial-num-value]      }
              {numeric-array-id (sub1[,sub2])             }
              {alpha-id$[length][=initial-str-value]      }
              {alpha-array-id$(sub1[,sub2])[length]       }

sub1, sub2   = numeric-expression which evaluates to a value in
               the range from 0 to 65535.

length       = numeric-expression which evaluates to a value in
               the range from 1 to 65535.  If length not
               specified, default to 16.
```

Discussion:

This statement declares a list of scalar variables that are added to the current function's RECURSIVE variable space. The variables may not already be declared as function private variables or parameters. Variables in the RECURSIVE variable space may be referenced by name only in the function body in which the declaration appears. Multiple declarations of variables in the function's private variable space are not permitted.

A new copy of a function's RECURSIVE variables is allocated and initialized each time the function is called and is released when the function RETURNs. The initial value of RECURSIVE variables is evaluated once only, at resolve time.

NOTE: Parameters of functions passed by value are always RECURSIVE variables.

Variables declared inside the body of any function without the STATIC or PUBLIC keywords are also by default RECURSIVE variables.

RECURSIVE variables explicitly declared outside the body of a function are not permitted.

DIM / RECURSIVE (cont.)

NOTE: All variables declared inside a function without the STATIC or PUBLIC keywords are RECURSIVE, and it is illegal to declare a RECURSIVE variable outside of all functions. Consequently, the RECURSIVE keyword is always optional and is usually only entered for clarity.

A DIM /RECURSIVE statement may not be executed in immediate mode.

The total allocated data for a function's recursive variables may not exceed 64K.

Examples:

```
0010 DIM /RECURSIVE Inited,FishName$16
0010 DIM /RECURSIVE names$(10)32, numbers(100)

0030 PROCEDURE ’do_nothing
   : ; local variables EXPLICITLY declared /RECURSIVE
   : DIM count, char$1, matrix(2,2), asciiCodes$(10)1
:    ENDPROCEDURE
```

Compatibility Issues:

This statement is supported only with Release IV or greater
LINs are supported on Release IV or greater.

References:

DIM /STATIC

General Form:

```
    DIM /STATIC dim-element[,dim-element]...
```

Where:

```
    dim-element = {numeric-id       [=initial-num-value]      }
                  {numeric-array-id (sub1[,sub2])             }
                  {alpha-id$[length][=initial-str-value]      }
                  {alpha-array-id$(sub1[,sub2])[length]       }

    sub1, sub2   = numeric-expression which evaluates to a value in
                   the range from 0 to 65535.

    length       = numeric-expression which evaluates to a value in
                   the range from 1 to 65535.  If length not
                   specified, default to 16.
```

Discussion:

The STATIC keyword at the start of a DIM statement indicates that all variables specified in the statement are private to a module or to the current function (if the statement is in the body of a function). Private variables may only be referenced by name from the same module (or function) in which the DIM statement declares them.

DIM /STATIC Within a Function Body

Each function may declare its own list of private variables. Each function private variable may be declared as either /STATIC or /RECURSIVE. A single copy of a function's /STATIC variables is allocated and initialized at resolve time. (By contrast, a function's /RECURSIVE variables are allocated and initialized each time the function is called). The initial value (if any) is computed only once, at resolve time.

Multiple declarations of variables in the function's private variable space are not permitted.

DIM / STATIC (cont.)

Variables in the function's private variable space may be referenced by name only in the function body in which the declaration appears. A function's private variable may be referenced by name anywhere in a function, provided the variable appears in a DIM STATIC statement previously in the function.

Variables declared inside the body of any function in a DIM statement without the STATIC or PUBLIC keywords are by default in the module's RECURSIVE variable list. In this case, the STATIC keyword is necessary if the same allocation must be used for iterative calls to the function.

DIM /STATIC Outside a Function Body

Each module may also declare its own list of private variables.

When a DIM STATIC occurs outside the body of all functions, the defined variables are private to the module.

If the variables are already declared in the module's private variable list, all attributes of the variable must agree with the previous declaration. In particular, strings must have the same length, arrays must have the same dimensions.

Variables in the module's private variable space may be referenced by name only in the module in which the declaration appears. A module's private variable may be referenced by name anywhere in a module, provided:

- The variable appears in a DIM STATIC statement previously in the module.
- When the reference occurs within the body of a function or procedure, no RECURSIVE or STATIC variable or parameter of the same name has been declared in the body of the current function (in which case it cannot be referenced in that function).

Variables declared outside the body of any function in a DIM statement without the STATIC or PUBLIC keywords are also by default in the module's private variable list. In this case, the STATIC keyword is not necessary but may be entered for clarity.

Memory for only one copy of a module's STATIC variables is allocated at resolve time.

DIM / STATIC (cont.)

Variable Reference Conflicts

If a variable identifier is declared in a module in a DIM /STATIC statement, and also in a DIM /PUBLIC statement (either explicitly or in a referenced PUBLIC section), this is not considered an error. Instead, separate variables are allocated in the private variable list and in the public variable list. A program reference to the variable refers to either the STATIC or PUBLIC variable, depending on which declaration occurred most recently in the program before the reference. To avoid confusion in such cases, it is advisable to place DIM /STATIC declarations as early as possible in the module, preferably immediately after any INCLUDE and USES declarations.

Similarly, if a variable identifier is declared outside all functions in a DIM /STATIC or /PUBLIC statement, and also inside a function with a DIM /STATIC or /RECURSIVE statement, this is not considered an error. Instead, separate variables are allocated in the module private or public variable list and in the function private variable list. A program reference in the function body to the variable refers to either the function variable or the non-function variable, depending on which declaration occurred most recently in the program before the reference. To avoid confusion in such cases, it is advisable to place DIM /STATIC declarations as early as possible in the function body, preferably immediately after the function header.

The result of a DIM /STATIC statements executed in Immediate Mode depends on the location of the next executable statement. If that statement is in the body of a function, the variable is private to the function. Otherwise, it is private to the module.

Examples:

```
0010 DIM /STATIC Inited,FishName$16,A(VectorSize)
0010 DIM /STATIC NameList$(20)45
0010 DIM /STATIC Ratio,E=EXP(1)
0010 DIM /STATIC FileName$8,Default_City$30="Moose Jaw"
0010 DIM /STATIC CornYields(_NUMBER_OF_MONTHS_KEPT_ON_RECORD)
0010 DIM /STATIC Buffer$(_MAX_OPEN_FILES)512
    :DIM /STATIC Temporary,Access$256
```

Compatibility Issues:

This statement is supported only with Release IV or greater

LIN’s are supported on Release IV or greater.

References:

$DISCONNECT

General Form:

```
     $DISCONNECT {ON [numeric-expression]}
                 {OFF                    }
```

NOTE: This statement is supported for Wang compatibility reasons only and its use in new development is not recommended.

The compiler generates a warning when this statement is encountered.

Discussion:

The syntax of this statement is supported only for compatibility with Wang 2200 Basic-2. An error 0 (not implemented) is returned if $DISCONNECT ON is executed. No operation is performed if $DISCONNECT OFF is encountered at execution time.

Examples:

Compatibility Issues:

This instruction, when executed on a Wang 2200 MVP, enables (ON) or disables (OFF) terminal disconnect detection.

Under NPL, terminal disconnect is not detectable.

This statement is syntactically recognized only by NPL Revision 3.0 or greater.

References:

DO/ENDDO

General Form:

```
DO [:statement] ... : ENDDO
```

NOTE: The use of this statement is not recommended. Refer to structured IF/ELSE/END IF as a better alternative.

Discussion:

The DO/ENDDO statements are used to specify a group of statements to be executed conditionally following an IF, ELSE, or ERROR statement. When the condition preceding the DO statement is true, all statements between DO and ENDDO are executed. If the condition preceding DO is false, then the statements between DO and ENDDO are not executed and program operation resumes with the statement following ENDDO.

DO/ENDDO statements must always be paired with ENDDO sequentially following DO in statement order. Improperly paired DO/ENDDO statements result in a P31 (Do not matched with ENDDO) error at execution time.

Examples:

```
0010 IF A=B THEN DO<
    :       C=D<
    :       E=F<
    :       ENDDO<
    : ELSE DO<
    :       G=H<
    :       I=J<
    :       ENDDO<
0020 PRINT A
```

In this case, if A=B, then only the statements C=D and E=F are executed. If A is not equal to B, then only the statements G=H and I=J are executed. In either case, the statements at line 20 are executed.

DO/ENDDO (cont.)

Nested DO Groups:

Nested DO/ENDDO groups are supported.

For example:

```
0010 IF A=B THEN DO<
    :    C=D<
    :    IF X=Y THEN DO<
    :          GOSUB 100<
    :          M=N<
    :          ENDDO<
    :    E=F<
    :    ENDDO<
    : ELSE DO<
    :    G=H<
    :    I=J<
    :    ENDDO<
0020 PRINT A
```

In this case, the statements GOSUB 100 and M=N are executed only if both A=B and X=Y are true. The statements C=D and E=F are executed whenever A=B, regardless of whether or not X=Y. As in the above example, the statements G=H and I=J are executed only when A is not equal to B.

Statements following DO, including ENDDO, may be on separate lines. During program execution, execution of an ENDDO when no DO has been executed does not cause an error. ENDDO actually performs no operation as a statement. Rather, it is used only at resolution time to determine the address of the next statement to execute when a condition preceding a DO statement is executed and evaluated as false.

For example:

```
0010 IF A=B THEN DO
0020      C=D
0030      GOSUB 100
0040      ENDDO
0050 X=C+1
0060 IF C<80 THEN 30
```

DO/ENDDO (cont.)

In this example, program operation may be transferred from line 60 to line 30 based on the value of C. Whenever this transfer takes place, the GOSUB 100 statement on line 30 are always executed and the ENDDO statement at line 40 performs no operation. It is only on the initial execution of this program, when line 10 is executed, and the evaluation A=B is false that the statements on line 20 and 30 are not executed.

NOTE: The programming technique demonstrated by this program, though valid, is not recommended. Programs which branch into or out of DO/ENDDO groups prove very difficult to maintain.

Use of DO/ENDDO in Nested Conditionals:

When a condition preceding a DO statement is not evaluated, statements following the DO statement are always executed.

For example:

```
0010 FOR X=0 TO 1<
    : FOR Y=0 TO 1<
    : PRINT "X=";X;"Y=";Y;<
    : IF X=1 THEN PRINT "A";<
    :          ELSE IF Y=1 THEN DO<
    :              PRINT "B";<
    :              ENDDO<
    : ELSE PRINT "C";<
    : PRINT<
    : NEXT Y,X

:RUN

X= 0 Y= 0 C
X= 0 Y= 1 B
X= 1 Y= 0 AB
X= 1 Y= 1 AB
```

In this example, whenever X=1, the condition Y=1 is not evaluated. Therefore, the DO statement is not executed. Therefore, the statement PRINT B is executed whenever X=1, regardless of the value of Y. Without the DO/ENDDO statements, "B" would not be printed when X=1 and Y=0. Because of the potential for confusion, use of DO/ENDDO in nested IF/THEN/ELSE constructs is not recommended.

DO/ENDDO (cont.)

Ambiguous Situations:

Placing ELSE clauses at the start of a new line can lead to ambiguous situations and is not recommended.

When ELSE is on a separate line from the IF statement it follows, in general, the ELSE statement is not executed even if the result of the IF statement is false. However, if the ELSE statement immediately follows an ENDDO statement, ELSE is executed (if the result of IF is false) even when ELSE is on a separate line.

For example:

```
0010 IF A=B THEN DO
0020     C=D
0030     E=F
0040     ENDDO
0050 ELSE X=Y
```

In this case, the statement X=Y is executed whenever A is not equal to B.

```
0010 IF A=B THEN C=D
0020 ELSE X=Y
```

In this case, the statement X=Y is never executed even when A does not equal B.

Another ambiguous situation occurs when DO/ENDDO is used in conjunction with the ERROR statement.

For example:

```
0010 DATA LOAD BAT/D11,(X)X$<
    : ERROR DO<
    :     E=ERR
0020      PRINT "ERROR ";E;" OCCURRED"<
    : ENDDO
0030 A=B
```

If DO/ENDDO was not used and an error did not occur on the DATA LOAD statement, program operation would resume at line 20 (the next line number following the ERROR statement). However, the use of DO/ENDDO changes this logic so that if no error occurs, program operation resumes at line 30 (following the ENDDO statement).

DO/ENDDO (cont.)

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

Nesting of DO/ENDDO is not supported in Wang 2200 Basic-2.

In Wang 2200 Basic-2 Revision 3.0 or higher, ELSE may be on a different line from IF, regardless of whether or not ELSE follows a DO Group (although, in prior releases, of Wang Basic-2, ELSE is always invalid if it is on a separate line from IF). In NPL, ELSE on a separate line from IF is only valid when it follows a DO Group. Refer to ELSE for further details.

References:

ELSE
ERROR
IF/THEN
IF/ELSE/END IF

DSC Alpha-operator

General Form:

```
alpha-receiver = [...] DSC alpha-operand [...]
```

Where:

```
alpha-operand = {literal-string  }
                {alpha-variable  }
                {ALL function    }
                {BIN function    }
                {system-variable }
```

Discussion:

The DSC (decimal subtract with carry) alpha-operator subtracts the decimal value of the alpha-operand from the decimal value of the alpha-receiver. The DSC alpha-operator may only be used in an alpha-expression in an alpha-assignment statement.

The DSC operation assumes that both operands contain valid, unsigned BCD (Binary Coded Decimal) data, where data consists of two digits per byte, and each digit is a number between 0 and 9. DSC does not check the operand contents for validity prior to subtracting; consequently, the resultant is unpredictable if operands contain invalid data.

Each byte of alpha-operand is subtracted (base 10 arithmetic) from each corresponding byte of the receiving alpha-variable; borrow propagation is automatically performed between bytes. The DSC operation is performed from right to left.

If the values of the alpha-operand and the receiving alpha-variable are of different length, then the DSC algorithm implicitly extends the shorter value with leading zeroes prior to the operation. If the resultant is larger than the receiving alpha-variable, then the extraneous high order bytes of the resultant are truncated before assignment.

NOTE: Contrary to conventional alpha-variable operations, the DSC alpha-operator operates on all bytes of an alpha-variable (either as a receiver or an alpha-operand), including trailing spaces.

DSC Alpha-Operator (cont.)

Example:

```
0010 A$=B$ DSC HEX(0001)
0010 A$=DSC STR(B$,5,3)

:0010 DIM A$3,C$3
:0020 PACK(######) A$ FROM 9990
:0030 C$=A$ DSC HEX(1298)
:0040 PRINT HEXOF(C$)
:RUN
008692
```

Compatibility Issues:

The Decimal Subtract with Carry operation accepts invalid packed decimal numbers as an alpha-expression in Wang 2200 Basic-2. In this case, the results are predictable but meaningless.

NPL is compatible with Wang 2200 Basic-2 with respect to the DSC function, provided the alpha-expression contains valid, packed decimal values.

References:

PACK
UNPACK
$PACK
$UNPACK
DAC
VER

DSKIP

General Form:

```
     DSKIP [file-number,]{numeric-expression[S]}
                         {END                  }
```

Where:

```
     numeric-expression = number of sectors or logical records to be
                          skipped.

     S                  = indicates that numeric-expression represents
                          physical sectors as opposed to logical re-
                          cords.

     END                = skip to end of file.
```

NOTE: The use of this statement is not recommended. Refer to Niakwa Data Manager as a better alternative.

Discussion:

The DSKIP statement is used with cataloged data files in order to set the "current" value in the Internal Device Table (for the file number specified) to a higher value. It permits skipping over logical records or physical sectors within the file.

If the END keyword is used, the new position of the "current" pointer is set to the end of the file. The offset of the end of file is stored in the file trailer sector each time a DATA SAVE DC END instruction is performed on the file. A DKSIP END statement also checks to ensure that the current pointer points to a valid end of file block.

If the END keyword is not used, the numeric expression indicates how far forward the current pointer should be advanced. If the letter "S" follows the expression, the value of the expression is a number of sectors which should be skipped. If the letter "S" does not follow the expression, the value of the expression is a number of logical records which should be skipped.

DSKIP (cont.)

When using the "S" parameter, the number of sectors specified is added to the current sector address. If this sector number would exceed the end of file, the current sector is set to the end of file sector address.

When not using the "S" parameter, the number of sectors to add to the "current" sector address is determined by actually reading forward through the specified number of logical records.

If an end-of-file record is encountered while skipping records, the skip operation terminates with the current sector pointing to the end-of-file record. The IF END condition indicates whether an end-of-file record was found.

Examples:

```
0010 DSKIP #1,5
0010 DSKIP 10
0010 DSKIP END
0010 DSKIP #Z,10S
0010 DSKIP #3,X*3
```

Compatibility Issues:

References:

IF END
Catalog Access Methods - Section 7.3.8 of the Programmer’s Guide

ELSE

General Form:

```
ELSE {simple-statement         }
     {DO [:statement]...: ENDDO}
```

NOTE: The use of this statement is not recommended. Refer to the structured IF/ELSE/END IF as a better alternative.

Discussion:

The ELSE statement is used to conditionally execute a simple-statement or DO Group which immediately follows an IF/THEN statement.

The ELSE statement can also be used following an ON x GOSUB or an ON x SELECT statement. In these cases, if evaluation of the ON x statement results in no action, the ELSE clause is executed. If evaluation of the ON x statement results in execution of one of the GOSUBs (or SELECTs), the ELSE clause is not executed.

When a single statement is executed when the IF/THEN condition is true, ELSE must be on the same line as the corresponding IF/THEN, otherwise a syntax error occurs.

HINT: As of Release IV, use of ELSE simple-statement is only permitted immediately following IF xxx THEN simple-statement or ON xxx GOSUB/GOTO/SELECT statements on the same program line. Other uses are flagged as a syntax error.

Examples:

```
0010 IF A=X THEN PRINT "A=X"
    :       ELSE PRINT "A DOES NOT = X"
0010 IF A=X THEN GOSUB 1020
    :       ELSE DO
    :            GOSUB 1030
    :            GOSUB 2000
    :        ENDDO
0010 IF A$=STR(B$,1,5) THEN PRINT "YES"
    :                  ELSE PRINT "NO"
0010 ON X GOSUB 100,200,300,400
    :      ELSE PRINT "ENTRY NOT ALLOWED"
```

ELSE (cont.)

Compatibility Issues:

Prior to Release IV, use of ELSE simple-statement was permitted in contexts other than immediately following IF xxx THEN, or ON xxx GOSUB/GOTO/SELECT statements, and resulted in the statement being ignored.

In Wang 2200 Basic-2 Revision 3.0 or higher, ELSE may be on a different line from IF, regardless of whether or not ELSE follows a DO Group (although, in prior releases of Wang Basic-2, ELSE is always invalid if it is on a separate line from IF). In NPL, ELSE on a separate line from IF is only valid when it follows a DO Group.

DO Groups are supported only in NPL Revision 3.0 or greater.

Programmers are advised to avoid the use of the ELSEDO/ENDDO constructs with the new IF/ELSE/ENDIF constructs of Release IV or greater.

NOTE: Future releases of NPL may restrict the use of ELSE DO entirely, with the new IF constructs. DO/ENDDO is still permitted.

References:

DO/ENDDO
IF/THEN
ON/GOSUB
ON/SELECT
IF/ELSE/END IF

ELSE Structured

General Form:

```
ELSE
```

Discussion:

The structured ELSE statement defines the start of the statements in an IF...ELSE...END IF structure which are executed only if the condition in the structured IF statement was false. It must be followed by an END IF statement, which indicates the end of the IF...ELSE...END IF structure.

It is possible to branch into the range of an IF...ELSE...END IF structure, although this is poor programming practice. If a structured ELSE statement of any kind is encountered during execution, control is transferred to the statement following the matching END IF statement.

Examples:

```
:IF X=Y
:  PRINT "SAME";
:ELSE
:  IF 'Fuzzy_Equal(X,Y)=0
:   PRINT "PRETTY MUCH THE SAME";X
:  ELSE
:   PRINT "DIFFERENT ENOUGH";X;Y
:  END IF
:END IF
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

IF Structured
END IF Structured

END

General Form:

```
END
```

Discussion:

The END statement is used to indicate the end of a program. END is an optional statement and may appear more than once in a program.

When END is executed, an "END PROGRAM" message and the amount of free space is displayed (program or variable space, the smaller of the two).

When END is executed under the interpretive RunTime, Immediate Mode is activated with the program and variables remaining in memory. If running under the non-interpretive RunTime, a colon appears on the screen. Press the EXECUTE key to exit NPL.

If an END statement is executed, program execution cannot be continued.

Examples:

```
1000 IF A=B THEN END
1000 END
```

Compatibility Issues:

To exit RTP or RTI, use $END.

References:

$END

$END

General Form:

```
$END [numeric-expression]
```

Discussion:

The $END statement causes an exit from the RunTime program. Unlike the END statement, no colon or free space listing appears. Program control is returned to the native operating system.

If a numeric-expression is specified, the termination code for the RunTime program is set to the value of the expression. Otherwise, the termination code is set to a value indicating normal completion (usually 0). Termination codes may be used to indicate why the RunTime program ended, or to indicate a failure to the calling program of the native operating system.

Examples:

```
0010 IF A$="Y" THEN $END
0010 $END
0010 $END X
```

Compatibility Issues:

This statement is supported only with Release 1.03 or greater.

This statement is not valid in Wang 2200 Basic-2.

Native operating system detection and legal values for $END termination codes vary, as does the value used to indicate normal completion. Refer to the appropriate NPL Supplement for specific details.

As of Revision 4.0 of NPL, a $END acts upon INCLUDEd modules by effectively deleting all discardable and non-discardable modules (i.e., even if the module has been modified and not saved or has common variables defined), to ensure /EXIT procedures are always executed to allow cleanup.

$END (cont.)

NOTE: Future releases of NPL may include an option to treat "non-discardable" modules differently at $END (i.e., automatic save of modified modules or a warning and option to save).

References:

END

END FUNCTION

General Form:

```
END FUNCTION ['identifier[$]]
```

Discussion:

This statement declares the end of the body of a function.

The optional identifier may be used for documentation purposes, and must match the corresponding FUNCTION name.

If execution "falls into" an END FUNCTION statement (i.e., no RETURN (value) statement is executed before the END FUNCTION is reached), an error is generated.

Examples:

```
0010 END FUNCTION 'Bessel
0010 END FUNCTION 'PrintableTime$
0010 END FUNCTION 'SubString$
0010 END FUNCTION                 : ;current
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

FUNCTION

END IF

General Form:

```
END IF
```

Discussion:

The END IF statement defines the end of the statements in an IF...ELSE...END IF structure. Refer to IF (structured) for an explanation of how this statement may be used in an IF...ELSE...END IF structure.

Examples:

```
:IF X=Y
:    PRINT "SAME";
: ELSE
:    IF 'Fuzzy_Equal(X,Y)=0
:         PRINT "PRETTY MUCH THE SAME";X
:    ELSE
:         PRINT "DIFFERENT ENOUGH";X;Y
:    END IF
: END IF
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

IF/ELSE/END IF

END PROCEDURE

General Form:

```
END PROCEDURE ['identifier]
```

Discussion:

This statement declares the end of the body of a procedure.

The optional identifier may be used for documentation purposes and must match the corresponding PROCEDURE name.

If execution "falls into" an END PROCEDURE statement (i.e., no RETURN statement is executed before the END PROCEDURE is reached), a RETURN is implied.

Examples:

```
0010 END PROCEDURE 'ProcessRecord
0010 END PROCEDURE 'Initialize
0010 END PROCEDURE 'Shutdown
0010 END PROCEDURE 'MoveWindow
0010 END PROCEDURE                :;current
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

PROCEDURE

END PUBLIC

General Form:

```
END PUBLIC [PackageIdentifier]
```

Discussion:

The END PUBLIC defines the end of a PUBLIC section of a module. There must be a matching END PUBLIC statement for each PUBLIC statement of a module. The PackageIdentifier (if any) of this statement must match the PackageIdentifier (if any) of the corresponding PUBLIC statement.

Examples:

```
0010 END PUBLIC
0010 END PUBLIC StringFunctions
0010 END PUBLIC StandardColorNames
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

PUBLIC

END RECORD

General Form:

```
END RECORD [record-identifier]
```

Discussion:

The END RECORD statement marks the end of the RECORD declaration for the indicated identifier. If no record-identifier is specified, the current record is assumed.

Once a complete record is declared, the user may declare instances of the record as string variables in DIM statements in which the length is specified by #RECORDLENGTH (record-identifier).

Examples:

```
0010 END RECORD Payroll
0010 END RECORD Employee
0010 END RECORD Passwords
0010 END RECORD
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

RECORD
#RECORDLENGTH
FIELD

END SWITCH

General Form:

```
END SWITCH
```

Discussion:

This statement declares the exit point of a numeric, string or logical CASE structure. Control is transferred to the statement following the END SWITCH statement when either the code of a matching CASE has finished executing, or no matching CASE was found.

Examples:

```
:SWITCH Widget_Type
:   CASE 0
:      PRINT "Gizmos"
:   CASE 1
:      PRINT "Thingammies"
:END SWITCH
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

CASE

ERR Function

General Form:

```
ERR
```

Discussion:

The ERR statement returns a unique two or three-digit error code of the most recent error condition.

Whenever a RunTime error is detected, ERR is set to the appropriate error code. Any reference to the ERR function resets the value to zero. ERR is also reset to zero whenever a RUN or CLEAR command is executed. Refer to Appendix B of the Programmer's Guide for a table of error codes.

NOTE: ERR is typically used in conjunction with the ERROR statement.

ERR may also be inspected to determine if overflow or other mathematical errors have occurred which are suppressed by the SELECT ERROR setting.

Additional information about errors may be obtained under program control by use of the ERR$ and $OSERR statements.

Examples:

```
0010 DATALOAD BAT (X)X$(): ERROR E=ERR: PRINT "ERROR";E;"OCCURRED"
0010 LIMITS T"FILE1",A,B,C,D: ERROR GOTO 100
0020 REM NO ERROR - NORMAL PROCESSING


0100 REM ERROR ROUTINE: X=ERR: IF X=48 THEN 110: IF X=93 THEN 120

0010 X=ERR : REM CLEAR ERROR
    :SELECT ERROR>69
    : Y=A/B+C/D
    :IF ERR>0 THEN PRINT "WARNING - Error has occurred!"
```

ERR Function (cont.)

Compatibility Issues:

The error codes returned by some I/O statements may be somewhat different from what is expected when executing in Wang 2200 Basic-2. In particular, disk access errors are commonly reported as I91 (Disk Hardware Error) or I90 (Disk Hardware Error) instead of some of the more esoteric of the error codes in the 90-99 range.

NOTE: Error trapping routines which require a specific error code to be returned in the event of an I/O error may be unsuccessful because of the above.

Three-digit error codes are supported only in NPL Revision 3.0 or greater and are not supported on the Wang 2200.

References:

CLEAR
ERROR
$OSERR
SELECT ERROR

ERR$

General Form:

```
     alpha-receiver = ERR$(error-code)
```

Where:

```
     error-code = a numeric-expression representing the error
                  code for which to return the error descrip-
                  tion. The expression must be in the range of 0
                  to 999 or an error results.
```

Discussion:

The ERR$ function returns a string of text which describes the specified error-code. The specified error-code must be a two-digit integer which matches one of the valid error-codes listed in Appendix B of the Programmer's Guide.

NOTE: **This function uses the ERR function to determine the error-code. ERR may also be used directly as the specified error-code when using ERR$.**

The text returned is identical to the literal message that appears when an untrapped error occurs in immediate mode. Text for error messages is stored in the file ERRORMSG.HLP with pointers to text stored in file ERRORMSG.IDX. These files are included with every RunTime package and should be installed in the NPL directory. Refer to the Supplement for further details on installation procedures.

NOTE: **Text returned by ERR$ should be used for information only. Text is subject to change in future releases or may be modified by the user.**

If, for any reason, these files cannot be accessed, or if the specified error-code is invalid, ERR$ returns all spaces.

Examples:

```
0010 $FORMAT DISK T/D10,
   :ERROR E=ERR
   :E$=ERR$(E)
   :PRINT "Error ";&E;" - ";E$;" occurred"
```

ERR$ (cont.)

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

On the Wang 2200, text for error messages is built into the operating system. It is, therefore, always found.

References:

ERR
ERROR
Error Codes - Appendix B of the Programmer's Guide

ERROR

General Form:

```
simple-statement:ERROR{simple-statement[:simple-statement]...}
                      {DO[:statement]...:ENDDO}
```

NOTE: The use of DO/ENDDO is preferred.

Discussion:

The ERROR statement is used to provide program control of recoverable errors. The following table lists all error-code ranges and indicates which are recoverable:

Error Code Range	Recoverable or Not
0-36	Not recoverable
37	Recoverable
38-47	Not recoverable
48	Recoverable
49-59	Not recoverable
60-99	Recoverable
100-199	Reserved
200-299	Extended NPL error codes - not recoverable
300-499	Extended NPL error codes - recoverable
500-599	External error codes - not recoverable
600-799	External error codes - recoverable
800-899	Reserved

ERROR (cont.)

When a recoverable error is detected in a simple statement which is immediately followed by an ERROR statement, the standard system error response is suppressed and execution continues with the statement following the ERROR verb.

When using the statement format of ERROR, any simple statements on a program line following an ERROR statement are executed only if an ERROR occurs. If a statement is followed by ERROR and the statement executes without an error, program execution continues with the first statement on the next line. Structured statements are permitted on the statement format of ERROR, but their use is not encouraged since breaking such a structured statement into multiple lines would mean that no error on the simple statement preceding ERROR could branch into the middle of a structured statement.

When using the DO Group format of ERROR, any statements within the DO group are executed only if an error occurs. If an error does not occur, program execution resumes with the first statement following the ENDDO statement.

Math errors which have been suppressed using the SELECT ERROR statement do not generate errors (default values are returned; refer to SELECT ERROR for details) and, therefore, are not detected by the ERROR statement.

For example, assuming a SELECT ERROR >65 has been executed, errors in the range of 60 to 65 cannot be detected by the ERROR statement.

Programs can detect the occurrence of the suppressed errors by use of the ERR function.

Examples:

```
0010 DATA LOAD DC OPEN T#2, "DATA"
  :ERROR GOSUB 800
  :PRINT "ERROR"
0020 DATA LOAD DC #2,X
```

In this case any missing file or I/O errors occurring in the DATA LOAD DC OPEN statement cause the subroutine at line 800 to be called, followed by the PRINT "ERROR" statement. If no error occurs on the DATA LOAD DC OPEN statement, execution proceeds at line 20 (the next line number).

```
0010 Q=T/W: ERROR DO: PRINT"W CONTAINS ZERO VALUE": Q=0: ENDDO: T=T+1
```

In this case, assuming that error 62 has not been suppressed by SELECT ERROR, the statement Q=0 is executed only if an error occurs on the statement Q=T/W but the statement T=T+1 is always executed.

ERROR (cont.)

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

On revisions of NPL prior to 3.0 and on the Wang 2200, a P37 error (Undefined Marked Subroutine) is non-recoverable.

Error codes of 100 or greater are generated only on NPL Revision 3.0 or greater.

References:

DO/ENDDO
ERR
ON ERROR
SELECT ERROR

EXEC Key

General Form:

```
EXEC (key)
```

Discussion:

The EXEC key performs two functions:

1. If in STEP Mode, the EXEC key executes the next program statement scheduled for execution, after which the program is again halted. This is intended as a convenient way of single-instruction stepping through program execution.

NOTE: If a STEP # statement has been executed to limit the normal debugging range, the halt is delayed until the program enters the specified range of lines.

2. If in Immediate Mode, but not in STEP Mode, pressing the EXEC key is the equivalent of entering the CONTINUE command, causing normal program continuation.

Pressing the EXEC key from Immediate Mode when a program is not resolved generates an error A09 - Program Not Resolved.

Examples:

Compatibility Issues:

The EXEC key provides an approximate equivalent to the HALT and CONTINUE keys on a Wang 2200.

The default key sequence for EXEC varies between different hardware versions of NPL.

References:

CONTINUE
STEP
Immediate Mode - Section 2.5 of the Programmer's Guide
Keyboard Equivalences - Appendix D of the Programmer's Guide

EXP Function

General Form:

```
EXP(numeric-expression)
```

Discussion:

The EXP function returns the value of the mathematical constant "e" (value 2.718281828459...) raised to a numeric-expression. This is valid wherever a numeric-expression is legal.

Examples:

```
0010 T = EXP(G3-7)
0010 C5(M3) = 10*EXP(F2(K))
0010 Z(3,J) = EXP(A4)/5
```

Compatibility Issues:

Due to the use of different algorithms, results of these functions may differ from functions evaluated in Wang 2200 Basic-2. In general, however, the functions are accurate to 13 significant digits.

References:

FIELD

General Form:

```
FIELD field-spec[,field-spec]...
```

Where:

```
field-spec       = {field-definition }
                   {/FILLER (fill-length)}


field-definition = field-identifier[[(dim1[,dim2])] ][=format-spec]
                                   [$[(dim1[,dim2])  ][len]]


format-spec      = {HEX(tsll)                   }
                   {<alpha-variable>            }
                   {string function-value       }
```

Discussion:

The FIELD statement declares a field type variable. The statement is only legal between a RECORD and END RECORD statement.

NPL associates each field type variable with a start, length and format of data within a record. Arrays also have associated dimension information.

NPL assumes that the start of the field is the position immediately following any previous FIELD and FIELD FILLER declarations.

The format-specification determines how NPL stores a field value in the record and the length in bytes of the field in the record. For arrays, the length is the length of each element). This must be a valid field format as supported by $PACK/ $UNPACK field (F=xx type) formats. If the format specification is not given, a default value is assigned. The default value for alpha fields is HEX(Annn), where VAL(HEX(0nnn)2) is the "len" of the string (or array element length). The default value for numeric fields is HEX(F108) (i.e., NPL internal numeric format).

FIELD (cont.)

The format-spec may also be an intrinsic string function, such as $FIELDFORMAT() or a user-defined string FUNCTION value (the FUNCTION must be declared in a previously INCLUDEd library module).

The user may enter the format-specification explicitly as a two-byte hexadecimal value HEX(tsll), where the "t" hexdigit specifies the field type, "s" specifies the subtype (or number of decimals) and "VAL(HEX(ll))" evaluates to the length of the field.

Alternatively, the user may enter the format-specification as an alpha-variable.

NOTE: FIELD declarations for string fields are permitted to specify an element or format-spec, but not both. For example:

```
FIELD Ticket $10 ; Legal
FIELD Ticket $ = HEX(A00A) ; Legal
FIELD Ticket $10 = HEX(A00A) ; Not Legal
FIELD Ticket $(10)2 = HEX(A002) ; Not Legal
```

The /FILLER field specification declares an unreferenceable section of a RECORD field type variable. The length expression indicates the number of bytes in the record that must be skipped.

Library functions are available which allow definition of the type, subtype and length as separate numeric expressions (which is not permitted with the hex format), using mnemonic CONSTANT values for the field type parameter.

Library functions are also available which allow definition of the format-specification as a data type and length using codes defined by the Niakwa Data Manager.

If the field format is to be the same as a previously defined field, the $FIELDFORMAT() built-in function may be used to indicate the field format.

All field-identifiers must be unique within the scope (STATIC/PUBLIC) specified.

FIELD (cont.)

Examples:

```
0010;
   : INCLUDE T/D13, "PCKFIELD"
   : INCLUDE T#_NPLDEV,"PCKFIELD"
   : USES PackFormats
   :;
   : DIM _BIN1$ = 'FieldType$( _PACK_UNSIGNED_BINARY_FORMAT,0,1)
   :;
   : RECORD Header
   :     FIELD Self_Id_Message$30
   :     FIELD /FILLER(64-1-30)      : ;next field at byte 64, 30 used now
   :     FIELD Info_Level=_BIN1$
   :     FIELD Min_Info_Level=_BIN1$
   :     FIELD Number_Sections=_BIN1$
   :     FIELD Screen_Size_Lines=_BIN1$
   :     FIELD Screen_Size_Columns=_BIN1$
   :     FIELD /FILLER(10)         : ;not interested in this
   :     FIELD Cursor_Position_Row=_BIN1$
   :     FIELD Cursor_Position_Col=_BIN1$
   : END RECORD
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

END RECORD
#FIELDLENGTH
$FIELDFORMAT
#FIELDSTART
$PACK
RECORD
#RECORDLENGTH
$UNPACK
RECORDs/FIELDs-Section 4.12 of the Programmer's Guide
PCKFIELDS-Section 3.4

String FIELD-Expressions - Alpha-Variable Equivalent

General Form:

```
alpha-variable-1.{field-identifier}$[(sub1[,sub2])]
                 {<alpha-variable-2>}
```

Where:

```
alpha-variable-1 =  the name of a buffer containing a
                    record.

field-identifier =  the name of a field in the record.

alpha-variable-2 =  an alpha-variable containing the name
                    of a PUBLIC string field.

sub1[,sub2]       = numeric expressions which are suscripts
                    to select an element of a string array
                    field.
```

Discussion:

String field-expressions are permitted wherever alpha-variables are allowed. The expression is equivalent to the substring of the record buffer, as defined by the string field name.

Examples:

```
0010 OldestChild$=Employee_Record$.Child_Name$(1)
0010 PRINT Input_Screen_Header$.Self_Id_Message$
0010 Alpha_Sort_Field$=Employee_Record$.<Selected_Field$>$
0010 IF Employee_Records$(I).<Selected_Field$>$=" "
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

String FIELD-Expressions (cont.)

References:

END RECORD
FIELD
$FIELDFORMAT
#FIELDLENGTH
#FIELDSTART
RECORD
#RECORDLENGTH

Numeric FIELD-Expressions - Term in Numeric Expression

General Form:

```
alpha-variable-1.{field-identifier }[(sub1[,sub2])]
                 {<alpha-variable-2> }
```

Where:

```
alpha-variable-1 =  the name of a buffer containing a
                     record.

field-identifier =  the name of a field in the record.

alpha-variable-2 =  an alpha-variable containing the name
                    of a PUBLIC numeric field.

sub1[,sub2]      =  numeric expressions which are sub-
                    scripts to select an element of a
                    string array field.
```

Discussion:

Numeric field-expressions are permitted as terms in any numeric expression. The term is equivalent to the unpacked value of the field in the record, as defined by the numeric field name.

Examples:

```
0010 Total= Total + PayrollRecord$.Federal_Withholding
0010 PRINT AT(InputScreenHeader$.Cursor_Position_Row,0);
0010 X=Employee_Record$.<Deduction_Name$>
0010 Total(I)=Total(I)+Employee_Record$.Miscellaneous_Deductions(I)
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

END RECORD
RECORD
FIELD
#RECORDLENGTH

$FIELDFORMAT Function

General Form:

```
     $FIELDFORMAT {field-identifier[$][()]}
                  {<alpha-variable>       }
```

Where:

```
     alpha-variable = an alpha-variable containing the name of a
                      PUBLIC field.
```

Discussion:

The $FIELDFORMAT intrinsic function returns the two-byte field specification for a given field identifier variable.

The $FIELDFORMAT function permits indirect specification of a PUBLIC field name using the same syntax as a field expression. The indirect specification <string> is permitted to have "$" or "()", or both, but these are ignored. The type of the field must be indicated by "$" or "()", or both, after the <string> specification.

Examples:

```
0010 BoxRowFormat$=$FIELDFORMAT(BoxRow)
0010 BoxTitleFormat$=$FIELDFORMAT(BoxTitle$)
0010 Employee_Name_Format=$FIELDFORMAT(Employee_Name$)
0010 Child_Name_Format$=$FIELDFORMAT(Childrens_Names$())
0010 Miscellaneous_Format$=$FIELDFORMAT(Miscellaneous_Deductions())
0010 $UNPACK(F=$FIELDFORMAT(BoxRow))STR(Rec$,#FIELDSTART(BoxRow))TO Row

0010 RECORD /PUBLIC MouseFace
   :   FIELD MouseNoseColor = HEX(B001)
   :   FIELD MouseWhiskersLength = HEX(B002 )
   :   FIELD MouseNickName$32
   :   FIELD MouseTail(20) = HEX(B004)
   : END RECORD
0020 ;
   : DIM _MaxFieldNameLength = 20
   : DIM F$2, N$_MaxFieldNameLength
   : ;
   : N$ = "MouseNoseColor"
   : F$ = $FIELDFORMAT(<N$>)
   : PRINT HEXOF(F$)     :; this prints B001
   : N$ = "MouseNickName"
   : F$ = $FIELDFORMAT (<N$>$)
   : PRINT HEXOF (F$)     :; this prints A020
   : N$ = "MouseTail"
   : F$ = $FIELDFORMAT(<N$>())
   : PRINT HEXOF(F$)     :; this prints B004
```

$FIELDFORMAT (cont.)

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

FIELD
$PACK
$UNPACK

#FIELDLENGTH Function

General Form:

```
     #FIELDLENGTH {field-identifier[$][()]}
                  {<alpha-variable>       }
```

Where:

```
     alpha-variable = an alpha-variable containing the name of a
                      PUBLIC field.
```

Discussion:

The #FIELDLENGTH intrinsic function returns the field length in bytes for a given field identifier variable.

The #FIELDLENGTH function permits indirect specification of a PUBLIC field name using the same syntax as a field expression. The indirect specification string is permitted to have "$" or "()", or both, but these are ignored. The type of the field must be indicated by "$" or "()", or both, after the <string> specification.

For example:

```
0010 RECORD /PUBLIC MouseFace
0020    FIELD MouseNoseColor=HEX(B001)
0030    FIELD MouseWhiskersLength=HEX(B002)
0040    FIELD MouseNickName$32
0050    FIELD MouseTail(20)=HEX(B001)
0060 END RECORD MouseFace

:X$="MouseNoseColor"
:X=#FIELDLENGTH(MouseNoseColor)     :; returns 1
:X=#FIELDLENGTH(<X$>)               :; <-- same using indirect
:X$="MouseNickName$"                :; $ is allowed but ignored
:X=#FIELDLENGTH(MouseNickName$)     :; Returns 32
:X=#FIELDLENGTH(<X$>$)              :; <-- same using indirect
:X$="MouseTail()"                   :; () are allowed but ignored
:X=#FIELDLENGTH(MouseTail())        :; returns 1
:X=#FIELDLENGTH(<X$>())             :; <-- same using indirect
```

#FIELDLENGTH Function (cont.)

NOTE: If the field identifier is an array, the length returned is the length of an element.

Examples:

```
0010 BoxRowLength=#FIELDLENGTH(BoxRow)
0010 BoxTitleLength=#FIELDLENGTH(BoxTitle$)
0010 Employee_Name_Length=#FIELDLENGTH(Employee_Name$)
0010 Child_Name_Length=#FIELDLENGTH(Childrens_Names$())
0010 Deductions_Length=#FIELDLENGTH(Miscellaneous_Deductions())
0010 G$=STR(Rec$,BoxRowStart,#FIELDLENGTH(BoxRow))
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

RECORD
FIELD
$FIELDFORMAT
#FIELDSTART

#FIELDSTART Function

General Form:

```
     #FIELDSTART  {field-identifier[$][()]}
                  {<alpha-variable>       }
```

Where:

```
     alpha-variable = an alpha-variable containing the name of a
                      PUBLIC field.
```

Discussion:

The #FIELDSTART intrinsic function returns the starting STR() position in a record for a given field identifier variable (a value of 1 means start of record).

The #FIELDSTART function permits indirect specification of a PUBLIC field name using the same syntax as a field expression. The indirect specification string is permitted to have "$" or "()" or both, but these are ignored. The type of the field must be indicated by "$" or "()", or both, after the <string> specification. For example:

```
0010 RECORD /PUBLIC MouseFace
0020 FIELD MouseNoseColor=HEX(B001)
0030 FIELD MouseWhiskersLength=HEX(B002)
0040 FIELD MouseNickName$32
0050 FIELD MouseTail(20)=HEX(B001)
0060 END RECORD MouseFace

:X$="MouseNoseColor"
:x=#FIELDSTART(MouseNoseColor)   :; returns 1
:x=#FIELDSTART(<X$>)             :;<-- same using indirect
:X$="MouseNickName$"             :;$ is allowed by ignored
:x=#FIELDSTART(MouseNickNames$)  :;Returns 4
:X=#FIELDSTART(<X$>$)            :;<-- same using indirect
:X$="MouseTail()"                :;() are allowed but  ignored
:x=#FIELDSTART(MouseTail(()      :;returns  36
:x=#FIELDSTART(<X$>())           :;<--same  using indirect
```

Examples:

```
0010 BoxRowStart=#FIELDSTART(BoxRow)
0010 BoxTitleStart=#FIELDSTART(BoxTitle$)
0010 Employee_Name_Start=#FIELDSTART(Employee_Name$)
0010 Child_Name_Start=#FIELDSTART(Childrens_Names$())
0010 Deductions_Start=#FIELDSTART(Miscellaneous_Deductions())
0010 $UNPACK(F=$FIELDFORMAT(BoxRow))STR(Rec$,#FIELDSTART(BoxRow))TO Row
```

#FIELDSTART Function (cont.)

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

RECORD
FIELD
$FIELDFORMAT
#FIELDLENGTH

FIX Function

General Form:

```
FIX(numeric-expression)
```

Discussion:

The FIX function returns the integer portion of the value of a numeric-expression, truncating the fractional portion of the value, if any. This is valid wherever a numeric-expression is legal.

Examples:

```
0010 Q=FIX(A)
0010 A(10)=30-FIX(Q)

:PRINT FIX(3.1)
 3
:PRINT FIX(3.9)
 3
:PRINT FIX(-8.1)
-8
:PRINT FIX(-8.9)
-8
```

Compatibility Issues:

References:

INT

FN Function

General Form:

```
FN {letter}(numeric-expression)
   {digit }
```

NOTE: The use of this statement is not recommended. Refer to FUNCTION as a better alternative.

Discussion:

FN is a general-purpose function which is used to invoke functions defined by the DEFFN function definition statement. A single, numeric argument must be passed when the function is used. The FN function may be used in any numeric-expression. This is valid wherever a numeric-expression is legal.

FN is used in conjunction with DEFFN. DEFFN defines the function while FN invokes the defined function.

NOTE: Other FN functions may appear in a DEF FN statement. FN() functions may be nested in this way up to five levels deep.

For example:

```
0010 DEF FNA(X)=FNB(X)+2
   : DEF FNB(Y)=FNC(Y)*5
   : DEF FNC(Z)=FND(Z)/3
   : DEF FND(T)=1+FNE(T)+2
```

This is legal, but if FNE() references other FN functions, evaluating FNA() generates an error P39 - FN's nested too deep.

Use of FN in Immediate Mode under NPL requires that the program be resolved in memory.

FN Function (cont.)

Examples:

```
0010 X = FNX(23)+45
0010 A,B,C=24+Y/FNR(24+FN4(W(10)))
0010 IF FNA(R) = FNA(R1) THEN 200

:0010 INPUT Y
:0020 X=FNB(Y)
:0030 PRINT Y,X
:0040 DEF FNB(A)=(A-3)/2
:RUN
? 9
 9                 3
:
```

Compatibility Issues:

Use of an FN function is not allowed in Immediate Mode on a Wang 2200.

References:

DEF FN
FUNCTION

FOR/BEGIN Structured

General Form:

```
FOR numeric-scalar=num-exp1 TO num-exp2 [STEP num-exp3]
BEGIN
```

Discussion:

The FOR/BEGIN statement defines the start of a FOR/BEGIN...NEXT structure. It may be followed by any number of statements, which comprise the body of the loop. It must then be followed by a NEXT statement with a matching numeric-scalar variable.

The FOR/BEGIN statement may be distinguished from the unstructured FOR statement by the presence of the BEGIN keyword at the end of the statement.

The numeric-scalar variable specified becomes the index-variable of the loop.

It is assigned the initial value specified by num-exp1. Num-exp2 (the target-expression) and num-exp3 (the step-expression) are evaluated once, at entry to the loop.

If the step-expression is positive, and the index-variable is greater than the target-expression, control is transferred to the statement following the matching NEXT statement.

If the step-expression is negative, and the index-variable is less than the target-expression, control is transferred to the statement following the matching NEXT statement.

Otherwise, the step-expression and target-expression are stored in an internal stack, and execution proceeds with the first statement in the body of the loop.

NOTE: A step-expression of 0 always results in execution of the loop body exactly once.

FOR/BEGIN Structured (cont.)

Repeated branching out of a FOR/BEGIN...NEXT loop body without exiting the loop in an approved manner can result in a stack overflow. The following conditions clear the stack information created by a FOR/BEGIN statement:

- Exiting the loop at the matching NEXT statement
- Executing a BREAK statement to exit the loop
- Executing a NEXT statement for an outer FOR loop
- If the FOR loop was executed after entering a function or subroutine, executing a RETURN statement clears the information.

Unlike the unstructured version, which is permitted to have multiple (or no) NEXT statements, the end of the loop body of a FOR/BEGIN...NEXT loop is well defined. This permits the loop body to be skipped if the entry conditions indicate this should be done. Also, the LOOP statement may be used to skip to the NEXT statement of the FOR/BEGIN...NEXT loop, and BREAK may be used to skip past the NEXT statement of the loop (clearing loop information on the stack).

Examples:

```
0010 FOR X=1 TO Y BEGIN
    : ;Note that if Y is less than 1, the loop is not executed at all
    : NEXT X
0010 FOR Index=1 TO N BEGIN
0010 FOR Course=Soup TO Nuts BEGIN
0010 FOR GreekLetter=_ALPHA TO _OMEGA BEGIN
0010 FOR Century=1000 TO 1900 STEP 100 BEGIN
0010 FOR XValue=0 TO 1.00 STEP .01 BEGIN
0010 FOR T=1 TO 20 BEGIN
    : Address$=$DET(T)
    : IF Address$=" " THEN BREAK
    :  Device$=$DEVICE(Address$)
    :  PRINT Address$;Device$
    : NEXT T
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

BREAK
LOOP
NEXT

FOR/TO

General Form:

```
FOR index-variable = numeric-expression1 TO
      numeric-expression2 [STEP numeric-expression3]
```

NOTE: The use of this statement is not recommended. Refer to FOR/BEGIN as a better alternative.

Discussion:

The FOR/TO statement is used in conjunction with the NEXT statement to create iterative loops during program execution. Each FOR/TO statement must be paired with a matching NEXT (or NEXT CLEAR) statement later in the program. Both FOR/TO and NEXT statements must specify the same index-variable.

The FOR/TO statement is used when a group of statements is to be executed repeatedly, while incrementing (or decrementing) an index-variable from an initial value (numeric-expression1) to a final value (numeric-expression2) by regular increments (numeric-expression3).

When first executed, the index-variable is set equal to numeric-expression1 and program execution continues with the following statement.

Upon execution of the corresponding NEXT statement, if a positive STEP value is used, the index-variable is tested to determine if the index-variable (+ STEP value) exceeds the final loop value. If a negative STEP value is used, the index-variable is tested to determine if the index-variable (- STEP value) is less than the final loop value. In either case, if the test is true, program execution continues with the program statement following the NEXT statement and the increment (or decrement) is not performed. If not, the index-variable is incremented (or decremented) and execution continues with the statement following the corresponding FOR statement.

NOTE: The statements between the FOR/TO and NEXT statements are always executed at least once, even if the initial value for the index-variable exceeds numeric-expression2.

FOR/TO (cont.)

Five rules should be considered when using FOR/TO loops:

1. The values of numeric-expression2 and numeric-expression3 are determined only once at the start of the loop.

2. No practical limit exists as to the number of loops which can be nested within a program.

3. Every FOR/TO loop that is encountered is executed at least once (even those with a STEP value of zero or those where the index-variable already meets the final value condition).

4. Branching into the middle of FOR/TO loops. The FOR/TO statement must be executed in order to enter a loop. If the FOR/TO statement has not been executed, execution of the corresponding NEXT statement generates an error P40 - No Corresponding FOR for Next Statement.

5. Branching out of a FOR/TO loop is allowed, although some caution should be used in doing so. Repeated branching from a loop without normal termination can fill the system stack where FOR/TO loop information is kept, causing a Stack Overflow error. If a loop is not terminated normally, the stack information is not cleared. There are several ways of clearing the FOR/TO stack information:

   - Setting the index variable equal to numeric-expression2 and then executing the corresponding NEXT statement.

   - Execution of a NEXT CLEAR statement.

   - If in a nested loop, executing the outermost NEXT statement clears inner loop stack information.

   - If in a subroutine, a RETURN or RETURN CLEAR command clears the FOR/TO stack information for loops within the subroutine.

FOR/TO (cont.)

6. Branching out of a FOR/TO loop with a RETURN is allowed. Execution of the RETURN statement automatically clears information from the system stack regarding all loops executed since the most recent GOSUB statement.

Examples:

```
0010 FOR I=1 TO 10: PRINT I: NEXT I
0010 FOR A=100 TO 10 STEP -5
0010 FOR X=N TO (T+1)*Z/R STEP S-2

0010 FOR J=1 TO 50
    : X=J*2/9
    : IF X=23.4 THEN J = 50 :;WHEN TRUE TERMINATE LOOP
0020 NEXT J
0030 END

:0010 X=10
    : FOR I=1 TO X
    :    PRINT I
    : NEXT I
:RUN
1  2  3  4  5  6  7  8  9  10
:
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

NEXT CLEAR is not supported on the Wang 2200.

References:

CONTINUE NEXT
NEXT
NEXT CLEAR

$FORMAT DISK

General Form:

```
    $FORMAT DISK T  [file-number,  ]
                    [disk-address, ]
                    [<address-var>,]
```

Discussion:

$FORMAT DISK is used to format disk media. The $FORMAT DISK statement performs differently depending on the device specified.

When a $FORMAT DISK statement is executed against a diskimage file, the file is deleted from the native operating system. The address still exists in the device table, but must be scratched using the SCRATCH DISK statement before it can again be accessed.

When $FORMAT DISK is executed against a "raw" diskette, the diskette is physically formatted in "raw" format.

Examples:

```
0010 $FORMAT DISK T/D32,
0010 $FORMAT DISK T/D10,
0010 $FORMAT DISK T#1,
0010 $FORMAT DISK TA$,
```

Compatibility Issues:

Refer to the NPL Supplement for details on "raw" diskette devices.

Use of the address-var parameter is supported only on NPL Revision 3.0 or greater and is not supported on the Wang 2200.

References:

SCRATCH DISK
Native Operating System Files as diskimage files - Section 7.3.4
Native Operating System "raw" devices as diskimage files - Section 7.3.5
Diskimage Files - Section 7.3.4 of the Programmer's Guide
"Raw" Devices - Section 7.3.5 of the Programmer's Guide

FUNCTION

General Form:

```
      FUNCTION'name return-type [(parameter[,parameter]...)]
                                    [attribute            ]...
```

Where:

```
     name         = identifier

     return-type  = [$         ]

     parameter    = [/POINTER  ][_]variable [length]
                    [_]variable([dim1,[dim2]])

     attribute     = {/PUBLIC  }
                     {/FORWARD }
                     {/EXTERNAL}
                     {/BEGINS  }
```

Discussion:

This statement declares the entry point and type of a named function, and the parameters to that function (if any). If the /FORWARD keyword is not specified, statements following the FUNCTION statement define the body of the function. These must be followed by a matching END FUNCTION statement. Refer to Section 4.8 of the Programmer's Guide for more information.

Return Value Types of FUNCTIONs

Functions must return either a numeric or string value. A "$" after the function identifier indicates a string return value. Absence of a "$" after the function identifier indicates a numeric return type. Numeric function return values can be used wherever numeric constants are permitted. String function return values can be used wherever string literals are permitted.

Using FUNCTIONs as Terms in Expressions

Functions may appear in expressions of the appropriate type using the syntax for a function term.

FUNCTION (cont.)

For example,

numeric function term	'identifier [(parameters)]
string function term	'identifier$[(parameters)]

Parameters (if any) are evaluated from left to right, and all parameters are evaluated before the call is made. The parameters specified are passed to the function body, and execution proceeds with the first executable statement in the function. When the function body executes a RETURN(value) statement, the value specified is used (as if replacing the 'identifier[(parameters)] term), and evaluation of the expression containing the function reference term continues.

Return Value of FUNCTIONs

Return values of string-valued functions may refer to recursive variables. Space allocated to return values is managed internally by the RunTime and should be transparent to the program. Return values are released (when they are no longer referenced) at a number of points, including:

- Prior to any evaluation of SPACE functions, or
- Prior to any allocation of new memory, in particular, each time a FUNCTION is called.

A performance penalty occurs when operating with a number of unreleased return values.

For example:

```
0010 FUNCTION 'Dup$(Value$1,Count)
    : DIM Result$1000
    : STR(Result$,,Count)=ALL(Value$)
    : RETURN(STR(Result$,,Count))
    :END FUNCTION
0020 PROCEDURE 'RunsABitSlower(/POINTER _Arg1$,/POINTER _Arg2$)
    : ;return values cannot be released
    : END PROCEDURE
0030 'RunsABitSlower('Dup$("X",100),'Dup$("Y",100) )
```

Calls to functions and procedures are permitted from Immediate Mode.

FUNCTION (cont.)

Unlike Immediate Mode calls to subroutines using GOSUB or GOSUB’, there is no implied HALT before functions or procedures are executed.

If a function is called from Immediate Mode, then, even if the function is halted or STOPped for debugging, any Immediate Mode statements following the function are eventually executed when the function returns.

NOTE: **As a result of this change, if statements are entered after an Immediate Mode GOSUB(’) statement, they are also executed when the function RETURNs. This behavior is different from that of previous releases. On previous releases, statements after an immediate GOSUB(’) were never executed when the RETURN was executed.**

For example:

```
:PRINT "->";’WindowName$(TopWindow);"->"
->MainWindow<-
:GOSUB ’GetShorty:  PRINT "Result is ";X
Result is 22     <- immediate mode code executed after RETURN
  :
```

NOTE: **Unlike previous releases, an immediate mode GOSUB or GOSUB’ no longer does an implied HALT at the first statement.**

Examples:

```
0010 FUNCTION ’Bessel(X)
0010 FUNCTION ’PrintableTime$(/POINTER _AnyString$)
0010 FUNCTION ’SetSubString$(/POINTER Var$,Start,Length,Val$80)/FORWARD
0010 FUNCTION ’SubString$(/POINTER Var$,Start,Length,Val$80)/BEGINS
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

END FUNCTION
Refer to Section 4.8 of the Programmer’s Guide

’Function-name (...) Numeric-Expression Equivalent

General Form:

```
    'function-name[(argument[,argument]...)]
```

Where:

```
    function-name = {Identifier          }
                    {<alpha-variable>    }

    argument      = {numeric-expression  }
                    {<alpha-variable>    }
                    {literal-string      }
```

Discussion:

FUNCTIONs declared as numeric return types (no "$" string type indicator) may appear wherever a numeric expression is permitted by specifying the function Identifier preceded by "’" and followed by an argument list.

Refer to the general discussion of the NPL function interface in Section 4.8 of the NPL Programmer’s Guide.

Example:

```
0010 Y0='Bessel(X0)
0010 Start_Size='Minimum_File_Size
0010 PRINT 'Target(Sales$),'Accrued('Pension(Employee_Code$))
0010 CONVERT 'Id(RND(1)) TO Badge$,(#####)
0010 Y='<CallBackFunction$>(X)
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

Refer to Section 4.8 of the Programmer’s Guide

'Function-name$(...) Literal-String Equivalent

General Form:

```
     'function-name$[(argument[,argument]...)]
```

Where:

```
     function-name = {Identifier         }
                     {<alpha-variable>   }

     argument      = {numeric-expression }
                     {<alpha-variable>   }
                     {literal-string     }
```

Discussion:

FUNCTIONs declared as string return types ("$" type indicator in header) may appear wherever alpha-variables are permitted by specifying the function Identifier preceded by "'" and followed by "$" and an argument list.

Refer to the general discussion of function interface in Section 4.9 of the Programmer's Guide.

Example:

```
0010 Y0$='Title$(X0)
0010 Start_Name='Default_File_Name$
0010 PRINT 'ColorSet$(Foreground,Background);'Surname$(Salesman$)
0010 CONVERT STR('Id$(Password$),2,3) TO Count
0010 VerifyFunction$='<CallBackFunction$>$(2)
```

Compatibility Issues:

References:

Refer to Section 4.8 of the Programmer's Guide

$GIO

General Form:

```
    $GIO [remark] [device-address,] (microcommand-sequence
                  [file-number,   ]
                  [<address-var>, ]

                            [,register-var]) [alpha-variable...]]
```

Where:

```
    remark                = a character string which identifies the
                            operation being performed.  The remark is
                            ignored at execution time, although only
                            letters, digits, and spaces are legal in
                            a remark.

    microcommand-sequence = {alpha-variable}
                            {hex-literal   }

    register-var          = an alpha-variable whose individual bytes
                            are used as registers to store control in-
                            formation.  Must be dimensioned length of
                            at least 10 bytes.

    alpha-variable        = an alpha-variable used for multiple char-
                            acter I/O operations, serving as a data
                            buffer.
```

Discussion:

$GIO is supported strictly for compatibility with Wang Basic-2 and is used for general input and output to the specified device, which must be a printer-type device (not a disk-type device). If no device is specified, the address used in the last SELECT TAPE statement is used.

$GIO (cont.)

The operation performed by the $GIO statement is determined by the contents of the microcommand-sequence. Each two bytes of this variable (or literal) defines a microcommand operation to be performed. Operations are performed sequentially, starting from the first two bytes, unless a branch microcommand is encountered. The available operations are detailed later in this document but, in general, consist of commands to transfer characters between the active device and either the register-var or alpha-variable buffer. Commands are referred to by the four hexadecimal-digit representation of the two bytes of the microcommand.

The alpha-variable specified as the register-var in the $GIO statement is used as a register variable. That is, certain bytes of the register-var variable are used as register bytes. The maximum readable length of a register variable is 15 bytes. All 15 bytes may be used, but certain bytes (5,6,8,9,10) are used by the system for status registers. The value of these registers may be changed by the system.

Register byte 8 is used to store error status information; register bytes 9 and 10 are used to store information related to character count during $GIO operation. The programmer should keep in mind that using these registers (5,6,8,9,10) is legal, but the system changes them at different times during the $GIO operation.

Control Code Status

When using the $GIO instruction, a special flag exists in memory called the "control code". The status of this "control code" is either "true" or "false", depending on the condition of the operation. Initially, the condition code is false.

Once the control code is set to true, $GIO operation is terminated unless the next instruction in the microcommand is one of two special branch instructions that check the control status. The special branch statements are:

Dxxx	(branch to xxx if control code true)
Exxx	(branch to xxx if control code false)

where xxx is a 3 hexadecimal-digit address within the microcommand sequence. The address is determined by the sequential position away from the first microcommand (which has address 000).

$GIO (cont.)

For example, an instruction such as "D004" would cause a branch to the fifth microcommand in the $GIO statement if the control code is true. If control code is false, the statement is ignored.

The following conditions set the control code to true:

1000	Set condition code true
15xy	If compare error bit is set
16xy	If complemented status code (register 8) and HEX(xy) HEX(00)
17xy	If status code (register 8) and HEX(xy) HEX(00)
1Bx	If during write operation buffer is empty
1By	If during read operation buffer is full
1Cxy	If x=y
1Dxy	If register pair x,x+1=register pair y,y+1
1Exy	If register x register y
1Fxy	If register pair x,x+1 register pair y,y+1

Microcommand Emulation

The following table lists the microcommands currently supported by the $GIO emulator, and the function provided by each.

2020	End of $GIO command string.
A000, A200	Print string from current alpha-variable buffer with no effect on TAB()
42r0	Print character in register r with no effect on TAB()
40xx	Print HEX(xx) with no effect on TAB()
0rHH	Set register r to HEX(HH)
44xx & 46x0	Send one-byte control sequence to device driver. Refer to the discussion below.
73x0	Select new I/O channel from reg x.
71hh	Select new I/O channel as HEX(hh).
870x	Read single bytes. Refer to the discussion below.
Cx20	Read multiple bytes into current alpha-variable buffer. Refer to the discussion below.
12x1	Set coarse timeout on next I/O operation (ignored).
12x2	Set fine timeout on next I/O operation (ignored).
1200	Disable timeouts (ignored).
760x	Status request (emulated only to screen address 05; responds appropriately for 80-column terminal).

$GIO (cont.)

860x	Wait for response (response equivalent to "ready").
Dxxx	Branch to location if $GIO condition code is "true".
Exxx	Branch to location if $GIO condition code is "false".
1A00	Set up next alpha-variable buffer for transmit.
18xx	Select alpha-variable buffer by number.
19rc	Increment decrement register (pair) #r.
1000	Set cc to TRUE.
14xx	Compare registers.
15xx	Compare registers.
16xx	Compare registers.
17xx	Compare registers.
1Cxx	Compare registers.
1Dxx	Compare registers.
1Exx	Compare registers.
1Fxx	Compare registers.
A604	Calculate and save LRC value in register 5. The LRC value is a cumulative XOR of all bytes in the data buffer.
7600	Check for application security code. Implementation of this feature is highly operating system-specific. Refer tothe appropriate NPL Supplements for implementation details.

Using $GIO Input (Cx20) Microcommands:

The Cx20 microcommands may be used to read bytes from a device address defined as a serial port. The TMO clause of the $DEVICE statement may be used to enable time-outs on read operations using the C620 microcommand so that the statement returns with zero bytes read if no bytes are present at the specified port. For the Cx20 series microcommands, the number of bytes actually read is returned in bytes 9 and 10 of the register-var variable of the $GIO statement. This capability provides for limited serial communications ability on systems where the complete asynchronous communications capabilities provided by the Niakwa Scientific and Communications Drivers (SCD) Package are not available.

$GIO (cont.)

Refer to Section 5.7 of the NPL Supplements for details on limited serial communications support and the availability of the SCD Package on your operating system. In addition, refer to Section 7.8 of the Programmer's Guide for further information on limited serial communications techniques. (Refer to $DEVICE for further information on the TMO clause.)

Reading Bytes Using 870x and Cx20 Type Microcommands

Another important use of these microcommands is in accessing native operating system files. These microcommands may be used to read single bytes (870x type, x not equal 0) or strings of bytes (Cx20, x=2,3,6 or 7 are all treated the same) from native operating system files. The 8700 microcommand (which, on Wang 2200 systems, means "read and discard one byte") must be used to "rewind" any file before data may be read from it. For the Cx20 series microcommands, the number of bytes actually read is returned in bytes 9 and 10 of the register-var variable of the $GIO statement. For the 870x microcommand, the timeout (HEX(10)) bit is set in byte 8 of the register-var variable if a read of length 0 (end of file) is returned.

For example, the following program reads and prints the "/config.sys" file.

```
0010 $DEVICE(/211)="/config.sys"
   : $GIO/211,(HEX(8700))        :REM REWIND THE FILE
0020 $GIO/211,(HEX(C620),G$) A$  :REM READ SOME BYTES
   : G=VAL(STR(G$,9),2)          :REM GET COUNT OF BYTES READ
   : IF G=0 THEN 30              :REM CHECK END OF FILE
   : $GIO/005,(HEX(A000)) A$,G   :REM PRINT THE BYTES TO SCREEN
   : GOTO 20
0030 $END
```

These microcommands may only be used for valid printer type devices (addresses /000, /204, /210, /211, ... /21F). Attempts to direct these microcommands to other device classes may result in a P48 (Illegal Device Specified) error or may result in no data being read.

Program logic which reads from native operating system files in the above manner should avoid polling the keyboard unless the HELP key has been disabled since the HELP key closes all native operating system files. Although continuing the program would reopen the files, the native operating system file pointer would be positioned at the end of the file (which is suitable for output, but not for input).

$GIO (cont.)

HINT: Only perform $GIO to print class devices which are $OPENed, or avoid using function calls to evaluate the alpha-variables in a $GIO statement.

For example, instead of:

```
0010 $GIO/01C(HEX(8700C620),L$)A$('NextRecordNumber)
```

use:

```
0010 R='NextRecordNumber
   : $GIO/01C(HEX(8700C620),L$)A$(R)
```

Examples:

```
:0010 M$=HEX(40414042)
    : $GIO /005,(M$)    :REM register-var, alpha-variable
                         not required for single
                         character output.
:RUN
AB (printed on screen)

:0010 X$="TEST"
    : $GIO /005,(HEX(A000))X$
:RUN
TEST  (printed on screen)
```

Compatibility Issues:

The A604 microcommand performs no operation on NPL releases prior to Revision 2.01.20.

The 7600 microcommand performs no operation on NPL releases prior to Revision 2.01.17 and is not supported on the Wang 2200.

The TMO clause of $DEVICE is supported only in NPL Revision 2.01 or greater.

Use of the address-var parameter is supported only on NPL Revision 3.0 or greater and is not supported on the Wang 2200.

Since, in general, the controllers attached to the Wang 2200 I/O bus are quite intelligent, the task of emulating the functions provided by this statement on a non-Wang 2200 system is formidable, to say the least.

Only a partial subset of the defined microcommands is implemented by NPL. The choice of microcommands has been dictated by the frequency of their usage and the feasibility of accurately reproducing their result.

$GIO (cont.)

Despite the large number of microcommands (65536, not all defined), the large majority of applications of $GIO use it as a patch to circumvent a few deficiencies in T-compatible BASIC and some or all releases of Wang 2200 Basic-2.

Support of microcommands is limited to those stated above. Use of $GIO to perform telecommunications is functional only if an appropriate device driver is installed. Refer to the Niakwa SCD manual for further details.

If function calls are used to evaluate an alpha-variable value in a $GIO statement, and the current device is not $OPENed, the current device of the $GIO is flushed before evaluating the function and reselected after evaluating. Consequently, if the $GIO is directed to a text file, and that file is closed for any reason while evaluating the function, the new file position is at the end of the file.

References:

SELECT TAPE

#GOLDKEY Function

General Form:

```
#GOLDKEY
```

Discussion:

The #GOLDKEY function returns a value between 0 and 65535 which is generated on a random basis for each RunTime Package diskette (Gold Key numbers are different). The #GOLDKEY function returns the same value for the interpretive runtime and the non-interpretive runtime on any one Gold Key. The intent of this function is to provide a method of copy protection for application software authors. This is valid wherever a numeric-expression is legal.

The following is a suggested procedure for implementing copy protection:

1. The application system devises an arbitrary hashing algorithm which we denote by FNA. This function produces a unique number for each argument in the range 1-65535, preferably in such a way that the algorithm is not obvious. In general, this number may be up to 13 digits long.

2. The application system devises a second arbitrary hashing algorithm FNB. This function should take the 13-digit number produced by FNA and return another number, again in such a way that the algorithm is not obvious. The vendor should reserve space on the program diskimage (in a data file) for a single value of FNB for which the software is authorized.

#GOLDKEY Function (cont.)

3. At one or more critical points, the application software computes the value of the FNA and FNB functions, based on the value of the #GOLDKEY function, and compare this with the value on the program disk. If the numbers match, the program proceeds. Otherwise, an authorization check program should be called. The authorization check program should:

   A. Inform the user that the software is not authorized for use with the particular Gold Key diskette being used.

   B. Display the vendor's telephone number and explicit instructions on how to contact personnel able to authorize use of the software.

   C. Display the value returned by the FNA algorithm.

   D. Request an authorization number to be supplied by the vendor. At this point, the user must contact someone at the vendor's organization who:

      a. Can determine whether the user has the right to use the software and, if so,

      b. Has access to a utility program which computes and displays the FNB value based on the FNA number supplied by the user.

         Once this number is supplied to the user, the authorizing program does the following:

   E. Saves the FNB number in the reserved area to enable future use of the software by the same #GOLDKEY and proceed.

Additional Comments About #GOLDKEY

It is very inadvisable for programs to take any destructive actions based on what is decided to be a "breach" of authorized use of the software. In particular, this is true because the replacement Gold Key diskettes (provided as a result of upgrades or media failure) usually has a different value for the #GOLDKEY function.

#GOLDKEY Function (cont.)

NOTE: All programs which are capable of producing the FNB number should be compiled with the -OBJFORMAT SCRAMBLED option, or SAVEd with the ! scramble protect option for distribution, so as to inhibit inspection or modification of the FNB algorithm.

This function is provided as a convenience to licensees, to be used or not used as they see fit. Although due care has been taken in the implementation of this function, Niakwa disclaims all responsibility for the integrity or reliability of any and all copy protection systems based on the #GOLDKEY function.

HINT: It is recommended that production software be set up to refuse access if #GOLDKEY equals zero.

NOTE: The NPL demonstrator diskettes set #GOLDKEY to zero. This provides a mechanism to create demo versions of application software which operates only with demonstrator diskettes.

#GOLDKEY Determination

A new program, GOLDKEY.OBJ, was added to the Niakwa Development Package to allow developers to determine the #GOLDKEY number for any Niakwa Runtime based on the Gold Key serial number without having to open the RunTime Package. This program can be run as any other Niakwa program. When executed, the GOLDKEY program prompts for the Gold Key serial number as shown below.

```
Enter Serial Number (1 - 65535) to convert to #GOLDKEY:
```

Once the serial number is entered, the program returns the correct #GOLDKEY code number that is necessary for some application security programs.

Examples:

Compatibility Issues:

The #GOLDKEY function is supported only on RTP Revision 1.03 or later.

#GOLDKEY is not supported in the Wang 2200.

References:

GOSUB

General Form:

```
     GOSUB {line-number     }
           {statement-label }
```

NOTE: Functions and Procedures are a better alternative. Statement labels should be used as a better alternative to line numbers.

Discussion:

The GOSUB statement is used to begin a subroutine which begins at the line-number or statement-label specified. The line number or label must exist in the current module.

If the target line-number or statement-label is located in a function body, this statement must also be in the body of the same function.

If the GOSUB statement is located inside a function body, the target line-number or statement-label must also be located in the body of the same function.

The GOSUB statement may be used within a subroutine (i.e., subroutines may be "nested"). The maximum number of nested GOSUB calls is limited by the amount of available memory. Typically, up to 60 nested levels are allowed. Each GOSUB encountered places information onto the stack. This information is cleared upon execution of a RETURN, RETURN CLEAR, or LOAD statement.

Repeated execution of GOSUB without execution of a RETURN, RETURN CLEAR, or LOAD statement can result in a stack overflow error.

Program overlays (LOAD) remove all subroutine RETURN information from the stack.

GOSUB (cont.)

GOSUB is legal as an Immediate Mode command, but with the following restrictions:

- The program must be resolved.
- When the RETURN statement is executed, any statements following the GOSUB statement are executed, unless execution HALTs due to being in STEP mode, or due to a CONTINUE RETURN implied HALT.

NOTE: **If statements are entered after an immediate mode GOSUB(’) statement, they are also executed when the function RETURNs. This is different from previous releases. On previous releases, statements after an immediate GOSUB(’) were never executed when the RETURN was executed.**

For example:

```
:PRINT "->";’WindowName$(TopWindow);"<-"
->MainWindow<-
:GOSUB ’GetShorty: PRINT "Result is ";X
Result is 22   <- immediate mode code executed after RETURN
 :
```

NOTE: **Unlike previous releases, an immediate GOSUB or GOSUB’ no longer does an implied HALT at the first statement.**

Examples:

```
0010 GOSUB Reset
0020 GOSUB ProcessRecord
0030 GOSUB Code_1
:GOSUB InKey
```

Compatibility Issues:

On NPL releases prior to Revision 4.0, program execution halts at the first statement within the subroutine. Press EXEC or enter CONTINUE to continue.

GOSUB is not a valid Immediate Mode command in Wang 2200 Basic-2.

This statement is supported only with Release IV or greater.

References:

GOSUB’

General Form:

```
     GOSUB '{<num-exp>}[(argument [,argument]...)]
            {integer        }
            {name           }
            {identifier     }
            {<alpha-variable>}
```

NOTE: The use of this statement is not recommended. Refer to FUNCTION or PROCEDURE as a better alternative.

Discussion:

The GOSUB’ statement is used to execute a subroutine in a manner very similar to the GOSUB statement. However, in place of a line number, an integer value or DEFFN’ identifier is specified that relates to a corresponding integer value or DEFFN’ identifier on a DEFFN’ statement. Upon executing the GOSUB’, the program branches to the related DEFFN’ statement. When a subsequent RETURN statement is executed, the execution of the program transfers to the statement following the GOSUB’ that invoked the subroutine.

NOTE: As of Revision 3.0 or greater, the associated DEFFN’ may be defined in an external subroutine rather than in the NPL program. In cases where a given DEFFN’ is defined both internally and externally, the internal routine is executed. When neither an internal nor external routine is present, an error occurs. Refer to Mixed Language Programming in Chapter 16 of the Programmer’s Guide for further details on external subroutines.

Optionally, parameters may be passed to the specified subroutine. The parameters may consist of constants, variables, and expressions. Upon execution of the GOSUB’ statement, the parameters specified are automatically assigned to a corresponding list of variables in the related DEFFN’ statement. Care should be taken to ensure that numeric parameters are passed to numeric-receivers and alpha parameters are passed to alpha-variables or a RunTime error occurs. The number of arguments in the GOSUB’ statement must match the number of parameters in the corresponding DEFFN’ statement. (Refer to DEFFN’ for further details.)

GOSUB’ (cont.)

GOSUB ’, with parameters, is legal as an Immediate Mode command, but with the following restrictions:

- The program must be resolved.
- When the RETURN statement is executed, any statements following the GOSUB’ statement are executed, unless execution HALTs due to being in STEP mode, or due to a CONTINUE RETURN implied HALT.

A numbered PUBLIC DEFFN’ called by a GOSUB’ statement may be indirectly specified by a numeric expression within angle brackets (< >). The expression is evaluated and truncated to an integer, if necessary. The result must be a number in the range 0-65535. A named PUBLIC DEFFN’ called by a GOSUB’ may be indirectly specified by an alpha-variable within angle brackets (< >). The alpha-variable must contain a valid identifier. To avoid any possible ambiguity, when the target marked subroutine is specified indirectly, it must be declared as a PUBLIC DEFFN’ subroutine or as an external subroutine.

If the marked subroutine is not indirectly specified, the function must be defined either in the same module, as PUBLIC in a PUBLIC section, or as an external subroutine.

NOTE: If statements are entered after an immediate mode GOSUB(’) statement, they are also executed when the function RETURNs. This is different from previous releases. On previous releases, statements after an immediate GOSUB(’) were never executed when the RETURN was executed.

For example:

```
:PRINT "->";’WindowName$(TopWindow);"<-"
->MainWindow<-
:GOSUB ’GetShorty: PRINT "Result is ";X
Result is 22   <- immediate mode code executed after RETURN
 :
```

NOTE: Unlike previous releases, an immediate GOSUB or GOSUB’ no longer does an implied HALT at the first statement.

GOSUB’ (cont.)

Examples:

```
0010 ; declare ’MySub public so that it may be used indirectly
   : DEFFN’MySub(B,C)/PUBLIC /FORWARD
   : ;
0020 DIM X=100,Y=12,SubName$="MySub"
   : ;
   : GOSUB ’MyAdd(X,Y)     : ; call subroutine ’MyAdd directly
   : GOSUB ’<SubName$>(X,Y)   : ; call subroutine ’MySub indirectly
   : END
0030 ;
   : DEFFN’MyAdd(B,C)
   : A=B+C
   : PRINT A
   : RETURN
0040 ;
   : DEFFN’MySub(B,C)
   : A=B-C
   : PRINT A
   : RETURN
RUN
112
88

 DONE

0010 GOSUB’<X>
0010 GOSUB’<A(X)>
0010 GOSUB’<VAL(A$(X),2)>
0010 GOSUB’$NAMEOF(DEFFN’ MySub)(A$,B$)
0010 GOSUB’<x$>(A$,B$)
```

Compatibility Issues:

On NPL releases prior to Revision 4.0, program execution halts at the first statement within a GOSUB’ executed from immediate mode.

Use of more than 16 parameters is supported only in NPL Revision 3.0 or greater.

Use of GOSUB’s above ’255 is supported only in NPL Revision 3.0 or greater.

Use of GOSUB’s above ’255 is not supported on the Wang 2200.

Use of external subroutines is supported only in NPL Revision 3.0 or greater.

Use of external subroutines is not supported on the Wang 2200.

GOSUB’ is not a valid Immediate Mode command in Wang 2200 Basic-2.

Use of named subroutines and indirect references are supported only in NPL Revision 4.0 or greater.

GOSUB’ (cont.)

References:

DEFFN’
External Calls - Chapter 16 of the Programmer’s Guide

GOTO

General Form:

```
GOTO {line-number     }
     {statement-label }
```

NOTE: The use of this statement is not recommended because it is not structured. Use statement labels instead of line numbers as a better alternative.

Discussion:

The GOTO statement is used to unconditionally transfer program execution to a specified line-number or labeled statement. The line number or label must exist in the current module.

If the target line-number or statement-label is located in a function body, this statement must also be in the body of the same function.

If the GOTO statement is located inside a function body, the target line-number or statement-label must also be located in the body of the same function.

GOTO is legal as an Immediate Mode command and causes program execution to resume at the specified line-number when program execution is continued. When using GOTO in Immediate Mode, the following restrictions should be observed:

- The program must be resolved in memory.
- The line-number specified must be an existing program line-number.

Examples:

```
0010 GOTO 1000
0010 GOTO 7299
0010 GOTO 3333
0010 GOTO Reset
0020 GOTO ProcessRecord
0030 GOTO Code_1
:GOTO InKey
```

GOTO (cont.)

Compatibility Issues:

This statement’s statement-label option is supported only with Release IV or greater.

References:

HALT Key

General Form:

```
HALT (key sequence)
```

Discussion:

The HALT key is used to invoke Immediate Mode during program execution or a listing operation.

Pressing the HALT key during program execution invokes Immediate Mode at the completion of the current statement. The program remains resolved in memory; normal program continuation is allowed.

Pressing the HALT key during a listing operation stops the listing at the end of the current line and terminate the list operation.

The HALT key is not operational under the non-interpretive RunTime program.

Operation of the HALT key can be suppressed under program control by setting byte 13 of $OPTIONS system variable to HEX(01) (refer to $OPTIONS for details).

Examples:

Compatibility Issues:

The HALT key under NPL only invokes Immediate Mode; on the Wang 2200 the HALT key is also used to STEP through a program.

The HALT key is supported on NPL Revision 2.00 and higher of the Interpretive Run-Time (RTI) program.

Refer to the NPL Supplement for the keyboard specific HALT key sequence.

References:

$OPTIONS

$HELP

General Form:

Form 1:

```
$HELP=alpha-expression
```

Form 2:

```
alpha-receiver=$HELP
```

Discussion:

Form 1

Form 1 of the $HELP statement is used to store an eight-character HELP entry name in the $HELP pseudo variable. This HELP entry name may refer to a stand-alone HELP file or a HELP entry in a combined, indexed HELP file (refer to $HELPINDEX below). If referring to a stand-alone file, the HELP entry name must be a valid native file system filename. An extension of .HLP is assumed if extensions are permitted by the native file system.

Upon depression of the HELP key by the operator at runtime, application program execution is suspended, the contents of the screen is saved, and the current content of the $HELP variable is inspected by the HELP processor. The HELP processor first attempts to locate the specified HELP entry by searching the HELPINDEX file, if one exists. If this fails, the HELP processor treats the HELP entry name as a stand-alone filename and attempts to locate the HELP entry as a stand-alone file on disk.

Once the HELP entry is found, the information contained is displayed on the HELP screen. If the HELP entry cannot be found by either of the above methods, the message "NO HELP AVAILABLE" is displayed on the HELP screen.

The HELP information displayed can be varied, depending on which program of the application system is executing. Further, the information may be varied, depending on which portion of a program is executing. This capability allows for very specific instructions to the operator, depending on the exact circumstances.

$HELP (cont.)

Execution of the application program is resumed by executing the Leave Help option, at which point the application screen contents is restored to its original state (before HELP was invoked).

There are two components to implementing HELP screens for an application system. First, creation of the HELP FILES and, secondly, strategic placement of $HELP statements throughout the application programs. Refer to Chapter 11 of the Programmer's Guide for further details on $HELP.

Form 2

Form 2 is used to inspect the current contents of the $HELP system variable. Refer to Chapter 11 of the Programmer's Guide for details on use of HELP and indexed Help files.

Examples:

```
0010 $HELP="ARINF01"
0010 $HELP=F$&"001"
0010 A$=$HELP
0010 X$,Y$=$HELP
```

Compatibility Issues:

This statement is not valid in Wang 2200 Basic-2.

References:

$HELPINDEX
$HELP - Chapter 11 of the Programmer's Guide

$HELPINDEX

General Form:

Form 1:

`$HELPINDEX=`*`alpha-expression`*

Form 2:

`alpha-receiver``=$HELPINDEX`

Where:

`alpha-expression` `= a length of 50 characters.`

Discussion:

In addition to stand-alone HELP files, the RunTime program supports Indexed HELP files. Indexed Help files are essentially combined "stand-alone" Help files and are useful in that they are generally easier to maintain and require less disk space. In revisions of NPL prior to Release IV, $HELP could handle indexed HELP files containing up to 256 individual HELP entries. With Release IV of NPL. $HELP is capable of handling indexed help files with large number entries. There is no longer a built-in limit, but only 4K bytes of index (256 entries) are loaded and searched at a time. Subsequently, access to later index keys becomes progressively slower. Individual HELP entries in a Indexed HELP file are accessed by using a special HELP INDEX file which contains a listing of all HELP entries in the Indexed file along with a location (byte pointer) for each.

Refer to $HELP, Chapter 11of the Programmers Guide for details.

Form 1

Form 1 is used to assign the $HELPINDEX system variable the native operating system file-specification of the Indexed HELP file. Form 2 of the $HELPINDEX statement can be used to examine the contents of the $HELPINDEX system variable.

$HELPINDEX (cont.)

The Indexed HELP file and its associated index must have the same filename and be located in the same directory or equivalent native operating system file structure. If extensions are supported by the native operating system, they are differentiated by their extension: .HLP for the Indexed file, .IDX for the INDEX file.

Upon depression of the HELP key by the operator at runtime, the $HELPINDEX variable is inspected by the HELP processor. The filename contained is used to locate both the Indexed HELP file and its associated INDEX file. The INDEX file is first searched for the specific HELP entry (current contents of the $HELP system variable). If found, the Indexed HELP file is referenced and the text at the specified location is displayed. If an entry is not found in the HELPINDEX, or the HELPINDEX file is not found, the HELP processor searches for a stand-alone HELP file in the current directory using the $HELP filename.

Refer to $HELP, Chapter 11 of the Programmer's Guide, for a detailed discussion of the internal format of HELP files.

HINT: It is recommended that any program which modifies the value of $HELPINDEX save the original value in a variable and then restore $HELPINDEX to its original value before exiting the program. This ensures that the original value of $HELPINDEX is not lost.

Example:

This example illustrates how to store the value of $HELPINDEX at the beginning of a program, set it to a new value for the duration of the program, set up different individual HELP references, and restore $HELPINDEX to its original value before exiting the program:

```
:0010 DIM X$50
:0020 X$=$HELPINDEX         : REM SAVE ORIGINAL VALUE
:0030 $HELPINDEX="/HELP/AR" : REM SET COMBINED HELP FILE NAME TO FILE
                              AR.HLP IN DIRECTORY /HELP; INDEX TO
                              AR.IDX IN DIRECTORY /HELP
:0040 $HELP="CUSTNO"        : REM SET HELP ENTRY TO "CUSTNO"
:0050 LINPUT "PLEASE ENTER CUSTOMER NUMBER" -A$
:0060 $HELP="CUSTNAME"      : REM SET HELP ENTRY TO "CUSTNAME"
:0070 LINPUT "PLEASE ENTER CUSTOMER NAME" -B$
:0080 $HELP=" "             : REM BLANK $HELP BEFORE EXITING
:0090 $HELPINDEX=X$         : REM RESTORE ORIGINAL $HELPINDEX
:0100 LOAD RUN"START"       : REM EXIT PROGRAM
```

$HELPINDEX (cont.)

Compatibility Issues:

This statement is supported only with Release 1.03 or greater.

This statement is not valid in Wang 2200 Basic-2.

References:

$HELP
$HELPINDEX
$HELP - Chapter 11 of the Programmer's Guide

HEX Function

General Form:

`HEX(hh[hh..])`

Where:

h = hexadecimal digit (0-9 or A-F).

Discussion:

HEX-literals are a form of literal-string. HEX-literals provide a method of expressing any eight-bit character in a constant. This is achieved by expressing a given character or code by its two-digit hexadecimal equivalent (digits 0-9 or A-F).

HEX literals may be used anywhere that an alpha literal may be used. Consequently, HEX literals are often used to express literals which cannot otherwise be expressed as alpha literals--that is, literals which must contain a quotation mark, a carriage-return or codes that do not have a keyboard equivalent. HEX literals are also used where the natural expression of a constant is in hexadecimal, like device control codes for printers, the screen, etc.

For example:

```
:10 A$=HEX(034E49414B5741) :REM ASSIGN A 7 BYTE HEX LITERAL TO A$
:20 PRINT A$               :REM PRINT A$ TO THE SCREEN
:RUN
```

This example would clear the screen and print the characters "NIAKWA" on the screen. The hexadecimal representation of the control code to clear the screen is 03, which was assigned to the first byte of the variable A$. In the remaining bytes, the hexadecimal representation of "NIAKWA" is stored. Consequently, when these bytes are issued to the screen, the described action is effected.

HEX Function (cont.)

The same result could be achieved in a different way by the use of both types of literals and by specifying literals directly in the PRINT statement as follows:

```
:10 PRINT HEX(03);"NIAKWA"
:RUN
```

Since no keyboard equivalent exists for HEX(03), we represented it as a HEX-literal. Since the letters in "NIAKWA" are all represented on the keyboard, we can use an alpha literal to represent it. The result is the same as the previous example, and the program becomes more readable.

Examples:

```
0010 PRINT HEX(03) :REM Will clear the screen
0010 PRINT HEX(01) :REM Will home the cursor
0010 PRINT HEX(0A0A0A0D)          :REM Will move cursor down three lines and
                                  then perform a carriage return.
0010 X$=HEX(418142)
```

Compatibility Issues:

For programs compiled with versions of the compiler prior to Release II, no distinction is maintained between HEX literals and ASCII literals. Regardless of the original form of the statement, literals are decompiled either as HEX literals or ASCII literals, based upon the values contained in the literal. Literals with all characters in the range HEX(20) to HEX(7F), except for HEX(22) (double quotes (")) are displayed as ASCII literals. All other literals are displayed as HEX literals.

NOTE: As of Release II, the original form of literals may be maintained by use of the KEEPREMS Compiler Option or the $KEEPREMS system variable (used when entering program text).

References:

$KEEPREMS

HEXPACK

General Form:

HEXPACK *alpha-variable1 FROM alpha-variable2*

Discussion:

The HEXPACK statement is used to convert ASCII character strings of hexadecimal characters into their binary equivalent. Each pair of characters in alpha-variable2 is converted to a single character in alpha-variable1. Alpha-variable2 must contain only the characters 0-9 and A-F, or the characters HEX(3A) - HEX(3F) which are treated the same as hexdigits "A" - "F".

Trailing spaces in alpha-variable2 are ignored.

Examples:

```
0010 HEXPACK A$ FROM B$
0010 HEXPACK STR(B$,1,3) FROM STR(D$(),1,6)
0010 HEXPACK A$() FROM W9$()

:0005 DIM A$12,B$6
:0010 A$="123456789ABC"
:0020 HEXPACK B$ FROM A$
:0030 LIST DIM *
:RUN
DIM A$12
                   "123456789ABC"      HEX(3132 3334 3536 3738 3941 4243)
DIM B$6
                   "?4Vx/Ö"            HEX(1234 5678 9ABC)
```

Compatibility Issues:

References:

HEXUNPACK

HEXPRINT

General Form:

```
HEXPRINT alpha-variable [{;} alpha-variable]...[;]
                         {,}
```

Where:

```
, = specifies begin printing on a new print-line.

; = specifies no blank spaces between alpha-variables.

    A trailing semi-colon suppresses the trailing HEX(0D)(trail-
    ing line-feed).
```

NOTE: The use of this statement is not recommended. Refer to PRINT HEXOF( ) as a better alternative.

Discussion:

The HEXPRINT statement is used to print the hexadecimal value of one or more alpha-variables. All characters of an alpha-variable are displayed, including trailing spaces.

Examples:

```
0010 HEXPRINT A$,B$,C$;
0010 HEXPRINT B$;C$;L$(3),L$(4)
0010 HEXPRINT R1$,R2$,R3$,R$4;
```

Compatibility Issues:

References:

PRINT HEXOF()

HEXUNPACK

General Form:

```
HEXUNPACK alpha-variable1 TO alpha-variable2
```

Discussion:

The HEXUNPACK statement is used to convert the binary value of an alpha-variable to the hexadecimal character equivalents of that value. Alpha-variable2 must be at least twice as long as alpha-variable1. If alpha-variable2 is longer than required, the remaining bytes are not affected.

Examples:

```
:0010 DIM A$2,B$4
:0020 A$=HEX(B751)
:0030 HEXUNPACK A$ TO B$
:0040 PRINT "B$=";B$
:RUN
B$=B751

:0010 DIM R$16,S$8
:0020 R$=HEX(44A9B522C650D119)
:0030 HEXUNPACK STR(R$,3,4) TO S$
:0040 PRINT "S$=";S$
:RUN
S$=B522C650
```

Compatibility Issues:

References:

HEXPACK

#ID Function

General Form:

```
    #ID
```

Discussion:

The #ID function is used to return the CPU identification number of the host processor if available. The operation of #ID is extremely hardware-dependent. In the event that CPU serialization is not supported by the host processor or operating system, #ID returns a value of zero. #ID is typically used in multi-user networks to distinguish between users. This is valid wherever a numeric-expression is legal.

Examples:

```
    0010 X=#ID
```

Compatibility Issues:

Refer to the appropriate NPL Supplement for #ID values on particular hardware versions of NPL.

References:

#TERM
#PART
Multi-user Capabilities - Chapter 7 of the NPL Supplements

$IF

General Form:

```
$IF {OFF} [{file-number   }] line-number
    {ON }  {device-address}
```

Discussion:

The $IF ON/OFF statement tests the device-ready condition of the specified device and branches to the specified line-number if the device is ready ($IF ON) or not ready ($IF OFF). If no device is specified, the device defined by the last SELECT TAPE statement is used. Since the sensing of device-ready information depends on the native operating system, the accuracy of this statement under NPL is very limited. Provided the device is configured, the default action of the instruction assumes a "device ready" status. The following exceptions are worth noting:

/000 Although not configured, the null address is always "ready" (used by many programs to decide whether they are running on a Wang 2200T, which returns a "not ready" status).

/001 The keyboard address is ready only if there is a key currently buffered from the keyboard and the partition is operating in foreground.

/005 The screen address returns a status of 'ready' if the partition is operating in foreground. If the partition is operating in background, a status of "not ready" is returned.

Examples:

```
0010 $IF ON 100
0010 $IF ON #2,300
0010 $IF OFF #1,1000
0010 $IF OFF /001,100
```

$IF (cont.)

Compatibility Issues:

Since the sensing of device-ready information depends on the native operating system, the accuracy of this statement under NPL is very limited.

Use of $IF to determine whether or not the partition is operating in background is supported only on NPL Revisions 3.0 or greater and is highly operating system-dependent. Refer to the NPL Supplements for further details on background partition support on your operating system.

References:

SELECT TAPE

IF Structured

General Form:

```
IF logical-expression
```

Where:

```
logical-expression =  {cond [logical-operator cond]...}
                      {true}
                      {false}

logical-operator   =  {AND}
                      {OR}
                      {XOR}

cond                = {alpha-value        rel-op alpha-value}
                      {numeric-expression rel-op numeric-expression}

alpha-value         = {alpha-variable}
                      {string-literal}

rel-op              = { =   }
                      { >   }
                      { <=  }
                      { >=  }
                      { <>  }
```

Discussion:

The structured IF is used to execute a conditional branch to another program location based on either a true or false decision. The structured IF statement defines the start of an IF...ELSE...END IF structure. It may be followed by a number of statements which are executed if the logical condition is true. It may optionally be followed by a structured ELSE statement, and a number of other statements, which are executed if the logical condition is false. It must be followed by an END IF statement, which indicates the end of the IF...ELSE...END IF structure.

IF Structured (cont.)

Refer to IF/THEN for a detailed discussion of evaluation of condition and use of logical operators.

The reserved words TRUE and FALSE are permitted to replace any logical expression, and always evaluate to a true or false condition, respectively. These expressions are typically not used in IF/END IF structured constructs, but may be useful as exit conditions from structured WHILE...WEND, or REPEAT...UNTIL loops which is terminated by a BREAK.

The structured IF is differentiated from the unstructured one by the absence of the THEN keyword on the IF statement, and the absence of anything following the ELSE keyword on the ELSE statement. The structured forms may not be mixed with unstructured forms.

Therefore, the following example is illegal:

```
0239 IF Nice_Long_Variable_Name = 56 THEN DO  : REM unstructured
   :    PRINT Beta_Particle_Count;
   : ELSE : REM structured stmt used in unstructured IF=not allowed.
   :    Beta_Particle_Count = 'Get_New_Beta_Count()
   : ENDDO
```

Examples:

```
0010 IF Name$="Bobby"
0010 IF Number=666
0010 IF Number=666 AND Name$="Bobby"
0010 IF Number=666 OR Name$="Bobby"
0010 IF Number=666 OR Name$="Bobby" AND Day =_THURSDAY

0010 :IF X=Y
   :    PRINT "SAME";
   : END IF
0020 IF 'Fuzzy_Equal(X,Y)=0
   :   PRINT "PRETTY MUCH THE SAME";X
   : ELSE
   :   PRINT "DIFFERENT ENOUGH";X;Y
   : END IF
```

Compatibility Issues

This statement is supported only with Release IV or greater.

IF Structured (cont.)

References

BREAK
DO
ENDDO
ELSE
END IF
REPEAT
UNTIL
WHILE
WEND
Structured Programming - Section 4.2.1 of Programmer's Guide

IF/THEN

General Form:

```
      IF logical-expression THEN direction:ELSE {simple-statement       }
                                               {DO[:statement]...:ENDDO }
```

Where:

```
     logical-expression  = {cond [logical-operator cond]...}

     logical-operator   =  {AND}
                           {OR  }
                           {XOR}

     cond                = {alpha-value        rel-op alpha-value       }
                           {numeric-expression rel-op numeric-expression}

     alpha-value         = {alpha-variable}
                           {string-literal}

     rel-op              = { =   }
                           { >   }
                           { <=  }
                           { >=  }
                           { <>  }

     direction           = {statement                }
                           {line-number              }
                           {DO [:statement] ...:  ENDDO}
```

NOTE: The use of this statement is not recommended. Refer to structured IF/ELSE/END IF as a better alternative.

Discussion:

The IF/THEN statement is used to test conditions and conditionally execute the specified statement, DO group, or branch to the specified line number if the overall evaluation of the statement is true. If the statement is false, program execution continues with the next statement unless the specified direction was a DO group. In this case, if the statement is false, program execution continues with the statement following the ENDDO statement.

IF/THEN (cont.)

The general definition of a logical-expression is defined here, and is referred to in other statements.

Evaluation of Conditions

Operands for conditions must be of the same type (alpha or numeric). In evaluation of numeric conditions, the expression on either side of the relational operator is fully calculated before comparison. In the evaluation of alpha conditions, the following rules apply:

- Comparison is performed on a binary basis, byte by byte, starting from the left-most byte.
- If the alpha values are unequal in length, the shorter value is implicitly extended by spaces (HEX(20)) and the comparison proceeds for the full length of the longer value.

Use of Logical Operators

Multiple conditions may be specified, separated with the logical operators AND, OR, or XOR.

When multiple conditions are specified, evaluation of the statement is performed from left to right. As conditions and logical operators are encountered, a net "truth" flag in memory is updated. Once evaluation of all conditions and operators is complete, this "truth" flag is the result of the statement.

IF/THEN (cont.)

The "truth" flag is set to true following evaluation of a logical operator under the following conditions:

AND	The net "truth" flag as established by evaluation of all conditions and operators before the AND operator must be true, and the condition following the AND operator must be true.
OR	The net "truth" flag as established by evaluation of all conditions and operators before the OR operator must be true, or the condition following the OR operator must be true (or both).
XOR	Either the net "truth" flag as established by evaluation of all conditions and operators before the XOR operator must be true, or the condition following the XOR operator must be true, but not both.

For example:

```
10 A=1: B=1: C=1: D=1
20 IF A=1 THEN PRINT "TRUE ON LINE 20"
30 IF A=1 AND B=2 THEN PRINT "TRUE ON LINE 30"
40 IF A=1 AND B=2 OR C=1 THEN PRINT "TRUE ON LINE 40"
50 IF A=1 AND B=2 OR C=1 AND D=2 THEN PRINT "TRUE ON LINE 50"
60 IF A=1 AND B=2 OR C=1 OR D=2 THEN PRINT "TRUE ON LINE 60"

:RUN

TRUE ON LINE 20
TRUE ON LINE 40
TRUE ON LINE 60
```

On line 20, the evaluation of the condition A=1 is true so the entire statement is true.

On line 30, the evaluation of the condition A=1 is true and the net "truth" flag is set accordingly. However, since the condition following the AND (B=2) is false, the net truth flag is set to false following evaluation of the AND operator and, therefore, the statement is false.

On line 40, the net truth flag after evaluation of the segment:

```
IF A=1 AND B=2
```

is false. However, evaluation of the OR C=1 sets the truth flag to true since C=1 is true.

IF/THEN (cont.)

On line 50, the net truth flag after evaluation of the segment:

```
IF A=1 AND B=2 OR C=1
```

is true. However, evaluation of the AND D= 2 sets the truth flag to false since both the preceding net truth flag and the condition following the AND (D= 2) are not true.

On line 60, the net truth flag after evaluation of the segment:

```
IF A=1 AND B=2 OR C=1
```

is true. In this case, evaluation of the OR D= 2 sets the truth flag to true since, while the condition D= 2 is false, the preceding net truth flag was true.

Non-evaluation of Conditions

When logical operators are used, all specified conditions are not necessarily evaluated. If a determination of the overall truth of the statement can be made without evaluating a given condition, that condition is not evaluated. For example:

```
10 A=1: B=1
20 IF A=2 AND B=1/0 THEN ...
30 IF A=1 OR B=1/0 THEN ...
```

On line 20, once the condition A= 2 is evaluated as false the entire statement must be false. Therefore the condition B= 1/0 is not evaluated. On line 30, once the condition A= 1 is evaluated as true, the entire statement must be true and again the condition B= 1/0 is not evaluated.

This means that conditions may contain invalid numeric-expressions. An error only occurs if the condition containing the invalid expression is actually evaluated.

The ELSE Clause:

Optionally, the IF/THEN statement may also be followed by an ELSE clause. The ELSE clause may contain one statement or a DO group which is executed only if the preceding IF condition is false. If the IF condition is true, the ELSE clause is not executed. Statements subsequent to statements in the ELSE clause are not affected and are executed normally regardless of whether or not the ELSE clause is executed.

IF/THEN (cont.)

NOTE: When a single statement follows ELSE, the ELSE statement must reside on the same line as the IF/THEN statement. However, if a DO Group follows ELSE, ELSE may be on a separate line. (Refer to ELSE and DO/ENDDO for further details on this subject.)

Use of IF/THEN Statements in the ELSE Clause:

An ELSE clause may contain an IF/THEN statement which can be followed by another ELSE clause. In this case, the IF statement in the ELSE clause is only executed if the preceding IF statement is false. If the IF statement in the ELSE clause is not executed because the preceding ELSE is true, the subsequent ELSE clause is not executed. Otherwise, the subsequent ELSE clause is executed or not executed, based on the results of the IF statement in the ELSE clause.

For example:

```
10 A=1: B=1
20 IF A=1 THEN PRINT "A=1"
    : ELSE IF B=2 THEN PRINT "B=2"
    :ELSE PRINT "NEITHER WAS TRUE"

:RUN

A=1
```

The IF B=2 statement is not evaluated because the first IF statement was true. Therefore, the ELSE statement associated with IF B=2 is not executed. The message--"NEITHER WAS TRUE"--would be printed only if both IF statements were false.

Examples:

```
:0010 IF Q=79 THEN 150
:0020 GOTO 50
:RUN
```

The above sample code branches to line number 150 if the value of Q is equal to 79; if the value of Q is less than or greater than 79, the code branches to line 50.

IF/THEN (cont.)

```
:0010 IF X$="RENTAL" OR W(1)=100 THEN PRINT "ERROR"
:RUN
```

The above example prints the word "ERROR" if EITHER the alpha-variable X$ is equal to "RENTAL" or the value of W(1) is equal to 100.

```
:0010 IF AC THEN DO B=D: X=Y: ENDDO: ELSE DO B=E: A=C: ENDDO
:RUN
```

If the value of A is greater than the value of C, then B is set equal to the value of D and X is set equal to Y, and the ELSE clause is ignored. If A is not greater than C, then B is set equal to E and A is set equal to C.

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

References:

DO/ENDDO
ELSE

IF END THEN

General Form:

```
     IF END THEN {statement               }
                 {DO [:statement]...: ENDDO}
                 {line-number             }

                        [:ELSE {statement                  }]
                               {DO [:statement] ...:  ENDDO }
```

Discussion:

The IF END THEN statement is a special form of the IF/THEN statement used to test whether an end-of-file marker in a disk file was read on a previous disk read (DATALOAD) statement. Whenever an end-of-file marker is read, the end-of-file flag is turned "on". The IF END THEN statement tests for the end-of-file flag. If it was set "on", the END condition is evaluated as "true"; otherwise, the END condition is evaluated as "false". Program execution then continues according to standard IF/THEN logic. Refer to IF/THEN for details.

An end-of-file marker is created by the DATASAVE DC END statement. When this marker is read by a DATALOAD DC or DA statement, the IF END THEN flag is set "on".

The end-of-file flag is turned "on" when an end-of-file marker is read by a DATALOAD DC or DA statement, and turned "off" following a subsequent DATALOAD statement or IF END THEN statement. The CLEAR and RUN commands also reset the end-of-file flag.

Examples:

```
:0010 DATALOAD DC X$,Y$,Z$,J,K,L
:0020 IF END THEN 500
:0030 PRINT X$,Y$,Z$,J,K,L
:0040 GOTO 200
:0500 GOSUB '123
:RUN
```

If the end-of-file marker has been read, then program execution transfers to line 500; otherwise, the values of the variables X$, Y$, Z$, J, K and L are printed.

IF END THEN (cont.)

```
0010 IF END THEN 9999 : PRINT "More Information Available"
0010 IF END THEN 1000
0010 IF END THEN PRINT "File has been exhausted"
    :        ELSE GOTO 0120
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

References:

DO/ENDDO
ELSE
IF/THEN

IMAGE (%)

General Form:

```
    %[[ character-string ] [ image-spec ]]...
```

Where:

```
    character-string = a string of alpha characters.


    image-spec       = [ - ][ $ ][ # [,]...]... [.][ #... ][^^^^ ][ + ]
                       [ + ]                                   [ - ]
                                                               [++ ]
                                                               [-- ]
```

Discussion:

The IMAGE statement is a formatted template for printing literals and variables with the PRINTUSING statement. It consists of a line-number, followed by a percent sign (%), followed by a format specification. A location for a formatted variable within the image is signaled by one or more # characters, possibly with additional numeric punctuation characters.

Any combination of printable characters may be part of the IMAGE statement, inserted before and/or after the image-specs. Either a leading or a trailing sign may be used, but not both.

During printing, each image-spec is paired off with the next item in the PRINTUSING statement item-list. Alpha variables treat numeric punctuation as # characters. If the alpha-variable is too long, it is right-truncated. If more items are supplied than there are image-specs, the image-specs are reused from the beginning.

To use an IMAGE for a PRINTUSING statement, the IMAGE statement must be the first statement on a line (must immediately follow the line-number).

IMAGE % (cont.)

The "$", ","(comma), "."(decimal point) characters can be changed in the PRINTUSING output from an IMAGE statement by replacing bytes 4,5, and 6 respectively of the $OPTIONS system variable with the desired output characters. Replacing these characters is primarily used for foreign currency applications. Refer to $OPTIONS system variable for more details.

Examples:

```
:0010 %The balance in your account is $##,###.##
:0020 T=1234.56: R=54.39
:0030 PRINTUSING 10,T+R
:RUN
The balance in your account is  $1,288.95

:0010 DIM F$50
:0020 %You will be billed for ### cartons at $##.##/lb.
:0030 C=87: P=11.97
:0040 PRINTUSING 20,C,P
:RUN
You will be billed for  87 cartons at $11.97/lb.

:0010 Q=5049.6: W=2.01
:0020 %##### = $####.## at +#.##%
:0030 PRINTUSING 20,"TOTAL",Q,W
:RUN
TOTAL = $5049.60 at +2.01%
```

Compatibility Issues:

In Wang 2200 Basic-2, the IMAGE statement must occupy a statement line by itself, and not be combined with multiple statements on the same line. This is not a restriction in NPL.

References:

PRINTUSING
$OPTIONS

INCLUDE

General Form:

```
    INCLUDE T [file-number,   ]   filename  [TO module-name]
              [disk-address,  ]
              [<address-var>, ]
```

Where:

```
    filename      = { <alpha-variable>  }
                    { literal           }

    module-name   = { <alpha-variable> }
                    { literal          }
```

Discussion:

The INCLUDE statement statically loads the specified filename from the specified disk-image into a separate module at resolve time (if it is not already loaded) and resolves it (if it is not already resolved). The filename and module-name are evaluated once only, at resolve time.

Once a module has been INCLUDEd, its declared PUBLIC sections become available to the module which executed the INCLUDE statement. Named PUBLIC sections must be referenced by the USES statement.

Nested INCLUDE operations are permitted. If the INCLUDEd module also contains further USES and INCLUDE statements within its PUBLIC section, then these are also available to the "main" module.

The new module may optionally be given a name specified in the TO clause. If no explicit module name is given, the filename is used. The "main" module, containing the application’s mainline, has a blank module name (" "). Module names must be unique within the workspace. If a module with the given name has already been loaded, it is not reloaded. Although any string may be used to name a file, NPL requires that the module name be a legal identifier (i.e., alphanumeric, starting with a digit).

INCLUDE (cont.)

NOTE: A variable used as the address in an INCLUDE statement must be assigned a value in a DIM statement, or must be a constant variable, or a common variable.

If such a variable is just assigned a value in the program the INCLUDE will fail because the INCLUDE is performed at resolution time, before the variable will be assigned during program execution.

The "TO module-name" specification is frequently used to load interchangeable components into the same module name. For example, different modules that handle color or monochrome displays could each be loaded into a module named "DISPLAY".

If an INCLUDEd module declares a /MAIN procedure (for initialization purposes), then it is executed before any referencing modules are allowed to execute.

A module remains loaded as long as the INCLUDE statement is part of the resolved program code. If an INCLUDEd module is no longer referenced by any other module at the end of a resolution pass, it is deleted (there are exceptions: (1) when the module has COM variables, and (2) when the module has been modified and not saved). Values assigned to any /PUBLIC variables are no longer valid after the INCLUDEd module is unloaded.

Modules may be independently scramble-protected. The fact that one module is scramble-protected does not prevent debugging or development in other modules.

If it is required to list or edit an INCLUDEd module it may be selected with the MODULE command.

If the INCLUDEd module has an /EXIT function, that function is executed before the module is deleted.

Examples:

```
0010 INCLUDE T PlotDriver$
0010 INCLUDE T#2, "BANKFILE" TO "DataFileSpecs"
0010 INCLUDE T "SOURCEIO"
0010 INCLUDE T<NiakwaLibrariesDevice$>,"FIELDPCK"
```

INCLUDE (cont.)

"MAIN" Module

```
10 : A program that uses nested include modules
 : INCLUDE T "COMBO"
 : DIM G$#RECORDLENGTH(TeaTime)
20 PRINT 'Funct(A)      :; Public FUNCTION in COMBO1
 : GOSUB 'Blob          :; Public DEFFN' in COMBO2

 : PRINT Pubvar         :; Public variable in COMBO3
 : G$.Milk=1
```

Program "COMBO":

```
10 PUBLIC
 :   INCLUDE T"COMBO1"
 :   INCLUDE T"COMBO2"
 :   INCLUDE T"COMBO3"
 : END PUBLICINCLUDE (cont.)
```

Program "COMBO1"

```
10 PUBLIC
:   FUNCTION 'Funct(X)/FORWARD   :;functions...
 : END PUBLIC
```

Program "COMBO2"

```
10 PUBLIC
 : DEFFN'Blob/FORWARD           :;DEFFN's...
: END PUBLIC
```

Program "COMBO3"

```
10 PUBLIC
  : DIM Pubvar                  :;Variables...
  : RECORD TeaTime              :;including RECORDs and FIELDs
  :    FIELD Lumps=HEX(B001)
  :    FIELD Lemon=HEX(B001)
  :    FIELD Milk=HEX(B001)
  :    FIELD TeaCosyName$10
  : END RECORD
  : END PUBLIC
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

Modules - Section 4.10 of the NPL Programmer's Guide
USES
PUBLIC

INIT

General Form:

```
INIT ({hh             }) alpha-variable [,alpha-variable]...
     {alpha-variable }
     {literal        }
```

Where:

hh = two hexdigits (0-9, or A-F).

NOTE: The use of this statement is not recommended. Refer to ALL ( ) function as a better alternative.

Discussion:

The INIT statement is used to set all characters in one or more alpha-variables to a character specified as the first character of a variable or as 2 hexdigits.

The INIT statement performs the same function as ALL().

Examples:

```
:0010 DIM A$8
:0020 INIT("?")A$
:0030 PRINT A$
:RUN
????????

:0010 DIM B9$4
:0020 C$="ABCDEFGH"
:0030 INIT(C$)B9$
:0040 PRINT B9$
:RUN
AAAA

:0010 DIM D1$(2)2,D2$(10)1
:0020 INIT(FF)D1$(),D2$()
:0030 HEXPRINT D1$()
:RUN
FFFFFFFF
```

Compatibility Issues:

References:

ALL

INPUT

General Form:

```
INPUT [literal-string,] {alpha-variable   }[,{alpha-variable   }]...
                        {numeric-receiver }  {numeric-receiver}
```

Discussion:

The INPUT statement is used to prompt the operator to enter data during program execution. An optional message is allowed in the INPUT statement to instruct operator input.

When an INPUT statement is executed, the optional message is displayed at the current cursor position followed by a "?" prompt. The program is suspended at this point until the requested values have been entered. Data is sequentially assigned to variables in the order they are entered, with the "?" prompt appearing until all variables have been assigned. When all variables have been assigned, program execution continues.

Values can be entered in one of two ways: one at a time by entering the value and pressing RETURN, or more than one value at a time by using a comma as the variable delimiter. If an operator enters RETURN with no data entered at an INPUT prompt, execution of the INPUT statement is terminated and the remaining variable values are unchanged.

If entering string values which contain commas within the string, the string must be entered within quotation marks.

For example:

```
:0010 INPUT A$
    : PRINT A$
:RUN
?"Chicago, Illinois"
Chicago, Illinois
```

DEFFN ' subroutines numbered '0 to '31, or '126 through '127, can be executed while the system is waiting for a response to input, by pressing the defined special function key. However, when the RETURN statement in the subroutine is executed, the INPUT statement is reexecuted from the beginning and any values previously entered in response to the INPUT statement must be reentered.

INPUT (cont.)

Data can also be entered in an INPUT statement by using defined special function keys. Refer to DEFFN' Keyboard Input statement for details on entering character strings using special function keys.

If invalid data is entered, an error message is displayed and the "?" prompt reappears, allowing data to be reentered.

The RECALL key recalls the last data typed during an input operation (useful for repetitive data entry). Refer to Section 5.4 of the Programmer's Guide for details on Line Editor features.

Pressing the HELP key during an INPUT operation generates the HELP display. The INPUT operation is continued when program execution is continued (with variable values unchanged).

All the functions of the Line Editor are available to allow entry or correction of the INPUT line. Refer to Section 5.4 of the Programmer's Guide for details.

In addition, the initial mode of the Line Editor for an INPUT statement may be set so that entered data overstrikes or is automatically inserted into the entered line. Also, the operation of the INSERT key for an INPUT statement can be set so that it either inserts a single space or switches between insert and overstrike modes. These choices for these options are set using byte 44 of the $OPTIONS system variable. Refer to $OPTIONS for more details.

Examples:

```
0010 INPUT A,B,C$
0010 INPUT "INPUT A,B,C$",A,B,C$
0010 INPUT A$(I)
0010 INPUT STR(A$,3,3),B

0010 INPUT "Input Some Data",A,B,C
0020 PRINT "END"
:RUN
Input Some Data? 1
? 4
? 12
END

:RUN
Input Some Data? 1,4,12
END
```

INPUT (cont.)

Compatibility Issues:

Execution of marked subroutines with parameters during response to INPUT is supported by Wang 2200 Basic-2. NPL does not support this feature.

References:

DEFFN ’ Keyboard Input
DEFFN ’ Subroutine
$OPTIONS

INPUT SCREEN

General Form:

```
     INPUT SCREEN alpha-variable [,AT (x,y)][,BOX (r,c)]
```

Where:

```
     x = a numeric-expression specifying the starting row.

     y = a numeric-expression specifying the starting column.

     r = a numeric-expression specifying the number of rows to input.
         For any value r, r+1 rows are input.

     c = a numeric-expression specifying the number of columns per row
         to input.  For any value c, c+1 columns are input for each
         row input.
```

Discussion:

INPUT SCREEN is used to read the specified portion of the current NPL screen mapping area into the specified variable. The screen is read row by row, starting at the specified x,y coordinates for the specified number of rows and columns.

If AT values are not specified, the following defaults are used:

x - starting row = 0
y - starting column = 0

If BOX values are not specified, the following defaults are used:

r - number of rows = 24-x
c - number of columns = (w-1)-y

where "w" is the current screen width (normally 80 but may be 132 on some terminals).

INPUT SCREEN (cont.)

NOTE: Defaults for BOX values when AT values are not specified are (24,79) (assuming an 80-column width) which are identical to the largest possible values for a PRINT BOX statement on an 80-column screen.

The use of the "r" and "c" parameters corresponds to the way PRINT BOX works. If the "r" or "c" parameter, in conjunction with the x and y values, would cause areas beyond the current width or length of the screen to be accessed, a P34 (Illegal Value) error results.

INPUT SCREEN is primarily intended to be used in conjunction with PRINT SCREEN to temporarily save and then redisplay a portion of the screen. This capability allows new "pop-up" type features to be added to existing applications. Refer to PRINT SCREEN for a detailed example of this functionality.

INPUT SCREEN only recognizes information which has been displayed by NPL print class output statements directed to device address 05. Use of any other function that affects the screen display may result in incorrect data being returned by INPUT SCREEN. Such functions may include use of $SHELL, use of external routines which update the screen, use of third-party, stay-resident programs which output to the screen, or native operating system messages.

Volume of Information Returned:

The information returned by INPUT SCREEN to the specified alpha-variable consists of an 80-byte string containing header information about the display, followed by three sections containing the actual characters, attribute and box graphic information, and color attribute information, respectively. The length of each of these three sections is ((r+1)*(c+1)) bytes where "r" is the number of rows specified and "c" is the number of columns specified. Within each section, the bytes represent the character position being read. Characters are read from the screen row by row for the specified range and column by column within each row for the specified range. In cases where the last row is off the screen, the r+1 row contains useful information only for section 2. The three sections are stored contiguously with no delimiter. If the programmer needs to know the starting byte location for any section, it must be determined by calculation based on the specified "r" and "c" parameters. If the specified alpha-variable is too small to contain the information generated, no error occurs. The information generated by INPUT SCREEN is simply truncated to the last complete section.

INPUT SCREEN (cont.)

NOTE: No partial sections are ever returned.

For example:

```
INPUT SCREEN A$,BOX(10,10)
```

This inputs rows 0-10, columns 0-10 into variable A$. In this case, to store all returned information, A$ must be dimensioned to:

header information	80 bytes
section 1	(r+1)*(c+1) or 11*11= 121 bytes
section 2	121 bytes
section 3	121 bytes
total	443 bytes

INPUT SCREEN of a full standard-sized screen would require 6080 bytes (80 + 3*25*80) to store all information returned. However, if an application did not require the color attributes, 2000 bytes less would be required, and, if the application also did not require the video attributes/box-graphics, another 2000 bytes less would be required, thus reducing the required size to 2080 bytes.

NOTE: Where a screen width greater than 80 or a number of lines greater than 24 is supported, additional space is required.

Contents of the Information Returned:

As indicated above, INPUT SCREEN returns 80 bytes of header information followed by three sections of ((r+1)*(c+1)) bytes each. The specific contents are as follows:

Header Information (Bytes 1 to 80)

Bytes 1-29 - terminal ID message. This is arbitrarily set by NPL to "2236DE R03 19200BPS 8+0 (USA)" regardless of the terminal and communications parameters being used.

Bytes 30-78 - supplementary header information. This information is automatically used by PRINT SCREEN. The exact contents are as follows:

Byte	Contents
30-63	Reserved - all(00)
64	Information level of header. This field indicates the level of information returned by INPUT SCREEN in the header. The value for the 3.0 revision is HEX(00), but will increase in future releases as new information is added to the header fields (bytes 31-80). Applications that use newer information may check this byte to ensure that the information contained in the header corresponds with the RunTime revision in use.
65	Minimum information level of header. This field is used internally by the RunTime to determine whether a sufficient revision of RTP is in use for PRINT SCREEN of a given buffer. This value is HEX(00) on the 3.0 revision but may be modified in the future as new information is added to the header fields.
66	Binary number of valid display sections. This is determined based on the size of the receiver-variable specified in conjunction with the "r" and "c" parameters specified.
67	Binary screen size (lines). 24 on current revisions.
68	Binary screen size (columns). 80 or 132 on current revisions.
69	Binary value of AT row value (x).
70	Binary value of AT column value (y).
71	Binary value of BOX row value (r).
72	Binary value of BOX column value (c).

73	Current color for background/foreground:
	HEX(80) bit - reserved (=0)
	HEX(08) - reserved (=0)
	HEX(x0) - background color (x=0 to 7)
	HEX(0x) - foreground color (x=0 to 7)

74	Current color for perimeter and underline
	HEX(80) bit - 0=dim perimeter, 1=bright perimeter
	HEX(x0) - perimeter color (x=0 to 7)
	HEX(08) - reserved (=0)
	HEX(0x) - underline replacement color (x=0 to 7)

75	Video modes when enhanced mode selected (by HEX(0E)) For all bits, 0=off; 1=on HEX(40)bit - reverse video HEX(20) bit - blink HEX(10) bit - bright HEX(08)bit - underline Other bits are 0 and reserved.

Byte	Contents
76	Flags for video HEX(01) bit: 0 = current video mode turned off by HEX(0D) 1 = current video mode not turned off by HEX(0D) HEX(02) bit: 0 = current video mode is normal (not enhanced) 1 = current video mode is enhanced Other bits reserved and 0.
77	Alternate character set status HEX(00) - normal character set in effect HEX(02) - alternate character set in effect Other values reserved.
78	Cursor status HEX(00) cursor off HEX(01) cursor on steady HEX(02) cursor on blinking Other values reserved.
Byte 79	Binary cursor position (row)
Byte 80	Binary cursor position (column)

Section 1 - Character Information

Each NPL character present within the specified AT, BOX range is placed in section 1 of the alpha-variable. Characters are read from the screen mapping area row by row for the specified range and column by column within each row for the specified range. The character returned by INPUT SCREEN is the NPL hexcode (not affected by $SCREEN value).

In some cases (such as after using $SHELL), the character is "unknown" to NPL.

In these cases a value of HEX(00) is returned for the character.

If row r+1 is off the screen, this section contains all HEX(00).

INPUT SCREEN (cont.)

Section 2 - Video Attribute and Box-graphics Area

For each character read by INPUT SCREEN, a one-byte bit mapped code is placed in section 2. This code represents the video attribute status, box graphic segments present, and whether the character is from the normal or alternate character set. The code is structured as follows:

HEX(80) bit	Character is from the alternate character set
HEX(40) bit	Reverse video attribute is on
HEX(20) bit	Blink video attribute is on
HEX(10) bit	Bright video attribute is on
HEX(08) bit	Underline video attribute is on
HEX(04) bit	Left horizontal box graphic segment is present
HEX(02) bit	Right horizontal box graphic segment is present
HEX(01) bit	Vertical box graphic segment is present

When "character" box graphics are in use, the vertical box graphic segment actually represents the south vertical box graphic segment (character boxes use south and north vertical segments while "true" box graphics use a single vertical segment). When regenerating character boxes, PRINT SCREEN references the box graphic segment in the character in the row immediately above to determine whether or not a north vertical segment is required. If the character above contains a vertical segment, a north segment is assumed to be required.

NOTE: This technique does not guarantee 100% proper restoration of character boxes. Some anomalies may be present.

Row r+1 is used in this section to represent horizontal box graphics segments which appear below row "r". Other bits for row r+1 will be off.

Section 3 - Color Attribute

For each character read by INPUT SCREEN, a one-byte code is placed in section 3. This code represents the background/foreground color attribute for the character. The background attribute is stored in the high-order nibble and may have a value of HEX(0x) the HEX(7x). The foreground attribute is stored in the low-order nibble and may have a value of HEX (x0) to HEX(x7).

INPUT SCREEN (cont.)

NOTE: The HEX(80) and HEX(08) bits are currently unused and set to zero. These bits are reserved for future expansion and should not be used by the application.

The color attribute values are set based on the execution of the dynamic color attribute control sequence:

```
        HEX(02000605 0b 0f 0u 0p 0i 0F)
```

where the current value of "b" and "f" are the background and foreground color attributes. Refer to Section 7.3.18 of the Programmer's Guide for further details on the dynamic color attribute selection control sequence.

Color attribute values are set only when the dynamic color attribute selection sequence is active. Byte 22 of $OPTIONS must be set to a non-zero value for this to be true. In addition, use of color attribute selection must be supported on the monitor in use when INPUT SCREEN is executed. Refer to the appropriate NPL Supplement for further details on monitors supported for dynamic color attribute selection.

NOTE: If color is generated by any other method (such as by attribute replacement on EGA monitors), these colors are not recognized by INPUT SCREEN.

If color attributes are not in use, the contents of each byte of section 3 will be HEX(07), black background with white foreground.

INPUT SCREEN (cont.)

NOTE: Applications which use INPUT SCREEN to store information for later redisplay may find it useful to store other related information. This may include:

$SCREEN - contains current screen translation table.

$OPTIONS - several bytes in $OPTIONS affect screen output operations.

$MACHINE - contains information about the environment including terminal type and graphics capability.

$BOXTABLE - determines whether or not character boxes are used and the character set to be used for character boxes.

Examples:

```
    0010 INPUT SCREEN A$
    0010 INPUT SCREEN A$, BOX(5,10)
    0010 INPUT SCREEN A$, AT(3,20),BOX(5,10)
    0010 INPUT SCREEN STR(A$,,80)
    0010 INPUT SCREEN STR(A$,24,236), AT(B-A+1,C-D+1),BOX(3,12)
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

Several features of the NPL implementation of INPUT SCREEN are not supported in Wang Basic-2:

- The AT and BOX parameters are not supported.
- The supplemental information returned in bytes 30-78 is not returned on the Wang 2200.
- Color attributes are not supported on the Wang 2200. Therefore, information about color attributes is not returned by INPUT SCREEN.
- PRINT SCREEN is not supported on the Wang 2200.

On the Wang 2200, INPUT SCREEN is supported only with MXE terminal controllers. In NPL, the INPUT SCREEN operation is supported on all systems.

INPUT SCREEN (cont.)

On the Wang 2200, the terminal self-ID message is actually generated by the Wang 22x6 terminal and may vary from one terminal to another. It also varies based on communications parameters in use. In NPL, the self-ID message is a constant and is the same for all terminals and all configurations.

On the Wang 2200, INPUT SCREEN generates a status message to the terminal indicating progress of the command. In NPL, no message is generated.

References:

PRINT SCREEN
Screen Handling - Section 7.3of the Programmer's Guide

INT function

General Form:

```
INT (numeric-expression)
```

Discussion:

The INT function returns the integer (whole number) or non-decimal portion of a numeric-expression. For a non-integer value, INT returns the greatest integer which is still less than the original value. This is valid wherever a numeric-expression is legal.

NOTE: INT is also useful for computing a "ceiling" function -INT(-x).

Examples:

```
0010 Q=INT(A)
0010 A(10)=30-INT(Q)

:PRINT INT(3.1)
 3
:PRINT INT(3.9)
 3
:PRINT INT(-8.1)
-9
:PRINT INT(-8.9)
-9

:0010 R=5: S=6: T=4
:0020 IF INT((R+S)/T) THEN PRINT "ERROR": ELSE PRINT "OK"
:RUN
ERROR

:0010 INPUT "How many in a box",N
    : INPUT "How many items",C
    : PRINT "Requires";-INT(-N/C);"boxes"
:RUN
How many in a box? 10
How many items? 92
Requires 10 boxes
```

Compatibility Issues:

References:

$KEEPREMS

General Form:

Form 1:

```
$KEEPREMS = alpha-expression
```

Form 2:

```
alpha-variable = $KEEPREMS
```

Discussion:

$KEEPREMS is a one-byte system variable which is used in the interpretive RunTime (RTI) to control generation of p-code used to maintain REM (remark) statements, program text indentation, and the display format of literals.

Form 1:

Values may be assigned to the $KEEPREMS system variable using Form 1. These values have the following effects:

HEX(00) - Equivalent to compiler option KEEPREMS OFF

REMs are not retained.

Program indentation is not retained. Return-graphics are generated at the end of each statement.

The initial format of literals is not retained. In this event, literals which contain only the characters HEX(20) to HEX(7F) (except for HEX(22)) are displayed as quote literals. Literals with any characters outside of this range are displayed as HEX literals.

$KEEPREMS (cont.)

HEX(01) (the default) - Equivalent to compiler option KEEPREMS ON

REMs are retained.

Program indentation is retained. Return-graphics are not generated automatically at the end of each statement (though return-graphics are retained where entered via SHIFT/INSERT).

The format of literals is retained.

HEX(02) - Equivalent to compiler option KEEPREMS DEC

REMs are retained.

Program indentation is retained, but return-graphics are generated at the end of each statement (as with a value of HEX(00)).

The format of literals is retained.

$KEEPREMS affects the p-code generated when lines of program text are entered in Immediate Mode, or when the $OBJECT function is used. Depending on the value of $KEEPREMS, additional information is saved which can affect only the display format of source code generated by the de-compiler during program edit or listing operations or during generation of SOURCE text by the compiler (B2C) when p-code files are used as input. It does not affect p-code execution in any way. Values of $KEEPREMS may be HEX(00), (01) (the default), or (02).

$KEEPREMS affects the generation of p-code as program lines are entered from the line editor. It has no effect when programs are saved to disk.

$KEEPREMS (cont.)

The size of the p-code generated is affected by $KEEPREMS as follows:

- Retention of literal format requires up to one extra byte per literal.
- Retention of program indentation requires up to two extra bytes per statement.
- Retention of REMs requires as much space as needed to store the remark.

$KEEPREMS performs no operation on the non-interpretive RunTime Program.

Form 2:

The current value of $KEEPREMS may be examined using Form 2.

Examples:

```
0010 X$=$KEEPREMS
0010 $KEEPREMS=BIN(0)
0010 $KEEPREMS=HEX(02)
0010 $KEEPREMS=A$

:$KEEPREMS=HEX(00)
:10 REM Set values to zero: FOR I=1 TO 10: A$(I)=HEX(30): NEXT I
:$KEEPREMS=HEX(01)
:11 REM Set values to zero: FOR I=1 TO 10: A$(I)=HEX(30): NEXT I
:$KEEPREMS=HEX(02)
:12 REM Set values to zero: FOR I=1 TO 10: A$(I)=HEX(30): NEXT I
:LIST
0010 FOR I=1 TO 10
    : A$(I)="0"
    : NEXT I
0011 REM Set values to zero: FOR I=1 TO 10: A$(I)=HEX(30): NEXT I
0012 REM Set values to zero
    : FOR I=1 TO 10
    : A$(I)=HEX(30)
    : NEXT I
```

$KEEPREMS (cont.)

Compatibility Issues:

The $KEEPREMS system variable is not valid in Wang 2200 Basic-2.

The $KEEPREMS system variable is implemented in Revision 2.00 and greater of NPL.

References:

KEEPREMS Option - Section 14.15 of the Programmer's Guide

$KEYBOARD

General Form:

Form 1:

```
$KEYBOARD = alpha-expression
```

Form 2:

```
alpha-receiver = $KEYBOARD
```

Where:

```
alpha-expression/alpha-receiver = is a length of 576 characters.
```

Discussion:

This statement allows a NPL application program to modify (Form 1) or examine (Form 2) the current keyboard translation table. The $KEYBOARD system variable contains the 576 byte keyboard translation table currently in effect. The keyboard translation table is used to translate hex values for keys received from the actual keyboard to hex values expected by NPL programs.

This table translation is necessary since many keyboards do not have exact equivalences for some NPL required keys. The RunTime program contains built-in defaults for this table which should be adequate for most applications. Exceptions would be for the international character set or for applications which require use of the CLEAR or CONTINUE keys. Refer to Appendix D of the Programmer's Guide for hardware-specific default values.

A convenient method of modifying the keyboard translation table is provided by the EDKEYBOA utility. This utility may be used to create a disk file with the values to be loaded for keyboard translation. When the RunTime program is invoked, it looks for this file and, if found, replaces the built-in default translation table values with values from this file. Refer to Chapter 13 of the Programmer's Guide for details on the EDKEYBOA utility and the location of the file from which keyboard defaults are loaded.

$KEYBOARD (cont.)

Alternatively, a program may directly access and modify the keyboard translation table from within a NPL program by using the $KEYBOARD statement.

Organization of the Translation Table:

On all keyboards supported by NPL, the keys can be divided into two groups:

1. "Simple" keys which generate only a single code to a native operating system program when pressed.

2. "Complex" keys which generate multiple codes to a native operating system program when pressed.

For example, on a Wang PC, the "A", RETURN and BACKSPACE keys are simple keys, generating HEX(41), HEX(0D), and HEX(08), respectively. The "EXECUTE" and "CANCEL" keys are complex keys, generating HEX(1FC5) and HEX(1FE0), respectively.

The keyboard translation table consists of two main parts.

- Part 1 contains the replacement values for "simple" codes sent by the keyboard.

- Part 2 contains the replacement codes for "complex" codes sent by the keyboard.

Each of these two parts of the table is further broken down into two sub-sections.

- Section 1 consists of 32 bytes which are used on a bit (32*8=256 bits) basis to determine whether the replacement code is to be classified as "standard" or "special" when passed to the NPL program. When a code is classified as "special", it is treated as a NPL Special Function Key by the program. When a code is classified as "standard", it is treated as a standard keyboard key (e.g., A-Z, a-z, RETURN, BACKSPACE, etc.). A bit value of zero indicates that the code is to be classified as standard. A value of 1 indicates that the code is to be classified as special.

$KEYBOARD (cont.)

- Section 2 consists of 256 bytes which contain the actual replacement character to be sent.

Translation Table Layout

Part I - contains replacement values for "simple" keys.

	Section 1 - 32 bytes (from 1 to 32)
	Section 2 - 256 bytes (from 33 to 288)

Part 2 - replacement codes for "complex" keys.

	Section 1 - 32 bytes (from 289 to 320)
	Section 2 - 256 bytes (from 321 to 576)
Total	576 bytes

The table operates on the basis of relative position based on the HEX code generated by the key pressed. Key codes HEX(00) to HEX(FF) correspond to byte or bit 1 to 256, relative to the start of the relevant section. For example, if the key pressed generates a simple code HEX(41) (the letter "A"), the replacement value is located as follows:

1. Since the key is "simple", part 1 of the table is accessed.
2. Based on the code of HEX(41), the value in byte number 66 (VAL(HEX(41))+1) of section 2 of part 1 (byte number 98 overall) is sent to the NPL program.
3. Based on the value of bit number 66 (HEX(02) bit of byte number 9), which is zero, the code is classified as "standard".

$KEYBOARD (cont.)

Examples:

The following example changes the replacement value for the function key labeled "1/INDENT" on the Wang PC (which produces "complex" code HEX [80]) to "special" HEX(01) ('1) (the standard default is '0 ("special" HEX(00)).

```
10 DIM X$(576)1
20 X$()=$KEYBOARD              : REM FETCH CURRENT VALUES
30 STR(X$(),449,1)=HEX(01)     : REM SET BYTE 129 OF PART 2,
                                 SECTION 2, TO VALUE HEX(01)
40 $KEYBOARD=X$()              : REM IMPLEMENT MODIFIED TABLE
```

NOTE: In this example, it was not necessary to modify the bit masks in section 1 of part 2 since the key modified was previously defined as "special".

Also, the result of this example would be that both the function key labeled "1/INDENT" and the function key labeled "2/PAGE" is translated to special function key 1 ('1). No key would produce special function key zero ('0).

WARNING--Incorrect modification of the keyboard table can result in "hanging" the application because keys are no longer translated to values understood by the application. In particular, it is possible to map the keyboard in such a way that no key available generates HELP or, even worse, that HELP can be generated but EXEC and CANCEL are unavailable (so the system is stuck in HELP until it is rebooted).

NOTE: Keyboard mapping is done based on the values returned by the native operating system from standard function calls. If the native operating system does not distinguish between, for example, the shifted and unshifted states of a key, it is not be possible for NPL applications to do this either.

Compatibility Issues:

The $KEYBOARD statement is not supported in Wang 2200 Basic-2.

$KEYBOARD (cont.)

References:

CLEAR
CONTINUE
Keyboard - Section 7.4 of the Programmer’s Guide
Chapter 13 of the Programmer’s Guide
Appendix D of the Programmer’s Guide

KEYIN

General Form:

Form 1:

```
KEYIN alpha-variable [,,line-number]]
```

Form 2:

```
KEYIN alpha-variable,line-number,line-number
```

Discussion:

The KEYIN statement is used to receive a single character from the keyboard. The KEYIN statement always examines the input keyboard buffer for any characters previously entered but not processed.

There are two forms of the KEYIN statement:

Form 1

Form 1 of the KEYIN statement causes the program to wait for a key to be entered, if no key is present in the input buffer. The value of the key is stored in the first position of the specified alpha-variable. If no line-number is specified, program execution always continues with the next program instruction. If, however, a line-number is specified, then program execution continues with the next program instruction only if the key received is what is termed a standard key. If the key received is a Special Function Key, then processing continues at the specified line-number. This allows for special handling for Standard versus Special Function keys. Refer to Section 5.4 of the Programmer's Guide for details on Standard versus Special Function keys. This form of the KEYIN statement is the preferred method of accepting operator input because the use of processor time to poll the keyboard is minimized.

KEYIN (cont.)

Form 2

Form 2 of the KEYIN statement is often referred to as the "polling" KEYIN. When this form of the statement is executed, the keyboard buffer is examined for the presence of a key. If a key is present, the value of the key is stored, program execution continues at the first line-number, if a standard key was received, or the second line-number if a Special Function Key was received. If no character is present, the program does not wait for a character, but rather continues execution with the next statement. This form of the KEYIN statement is frequently used to clear the keyboard buffer in the event of an unexpected error condition so that operator type ahead is not processed inappropriately.

For example:

```
0010 KEYIN X$,10,10 : REM Clear Buffer
0020 PRINT AT(5,10); "Enter 'Y' or 'N'"
0030 KEYIN Y$
```

Individual polling KEYINs may also be used by long-running processes to periodically check for operator intervention (cancel, request for status report, HELP).
Pressing the HELP key in response to KEYIN generates the HELP display.

Examples:

```
0010 KEYIN R$
```

In the above example, program execution waits until a key is pressed. The key value is then stored in R$ and the next program statement is executed. It is not possible to distinguish between standard and special keys in this case.

```
0010 KEYIN F$(1),,50
```

In the above example, program execution waits until a key is pressed and places the keyin value in F$(1). If a special function key is pressed, program execution transfers to line 50, otherwise, the next program statement is executed.

```
0010 KEYIN J$,30,40
```

In the above example, the presence of a standard character in the keyboard input buffer causes the program to transfer to line 30, a special function key to line 40, and no input at all to the next program statement.J$ receives the key value, if any.

KEYIN (cont.)

Compatibility Issues:

The Wang 2200 allows KEYIN to be directed to a device other than the keyboard. The KEYIN statement only accepts keys from the keyboard in NPL. A syntax error is generated if a device address or file number is specified in the statement.

At runtime, if a SELECT INPUT statement has been executed to any device other than the keyboard, the KEYIN statement is still wait for input from the keyboard in NPL.

References:

LEN Function

General Form:

```
LEN (alpha-variable)
```

Discussion:

LEN is a numeric function which is used to determine the number of characters in an alpha-variable. All characters in the string are counted, including leading and embedded spaces, and ignoring all trailing spaces. This is valid wherever a numeric-expression is legal.

NOTE: If the variable contains all spaces, a value of 1 is returned.

When executing a LEN function of a STR function, the LEN function returns the defined length of the alpha-variable, e.g., trailing spaces are not ignored in this case.

To determine the dimensions of an alpha-array with one dimension under program control:

```
0010 N=LEN(STR(A$(1)))
   : M=LEN(STR(A$()))/N
```

Current dimensions of A$() are A$(M)N.

Examples:

```
0010 X=LEN(A$)
0010 X=LEN(A$())
0010 X=LEN(STR(A$,X))
0010 X=LEN(STR(A$()))

:0010 T$="EFGHIJ  "
:0020 PRINT LEN(T$)
:RUN
6
:0010 H$="EF GHIJ  "
:0020 PRINT LEN(H$)
:RUN
7

:0010 B$="     "
:0020 PRINT LEN(B$)
:RUN
1

:0010 A$="E FGHIJ   KLM   "
:0020 PRINT LEN(STR(A$,,16))
:RUN
16
```

LEN Function (cont.)

Compatibility Issues:

References:

LET Alpha Assignment

General Form:

```
     [LET] alpha-variable [,alpha-variable]... = alpha-expression
```

Where:

```
     alpha-variable    = {scalar alpha-variable}
                         {alpha array element}
                         {alpha array}
                         {STR() function}

     alpha-expression = {[alpha-operand][alpha-operator alpha-operand]...}
                        {alpha-operand  [& alpha-operand]...            }
                        {string  field-expression                       }

     alpha-operand     = {alpha-variable }
                         {literal-string }
                         {ALL function   }
                         {BIN function  }
                         {alpha  function}
                         {string  field-expression}


     alpha-operator    = { ADD[C]     }
                         { AND        }
                         { BOOLh      }
                         { DAC        }
                         { DSC        }
                         { OR         }
                         { SUB[C]     }
                         { XOR        }

     &                 = concatenation alpha-operator

     h                 =  hexdigit, 0-9 or A-F
```

LET Alpha Assignment (cont.)

Discussion:

The LET statement is used to assign the result of the evaluation of the expression on the right-hand side of the "=" to the variable or variables on the left-hand side of the "=".

NOTE: The null alpha-expression is not allowed.

There are two forms of the assignment statement: numeric assignment and alpha assignment. Type conversion is not performed and, therefore, mixed-type assignment generates an error. Refer to LET Numeric Assignment statement for details on how to assign values to numeric-receivers.

The use of the word LET, if omitted, is assumed.

Two Types of Alpha-Expressions

From the general form, it would first be noted that there are, in fact, two types of alpha-expressions, the sole significance of which is related to the concatenation operator (&). This operator provides the capability for concatenation of the value of two alpha-operands to form a single character string. This alpha-operator is treated specially and may not be combined with any other operator (except itself) in the same alpha-expression. The concatenation operator is discussed in CONCATENATION.

Evaluation of Alpha-Expressions

As defined by the general form, an alpha-expression may contain any number of alpha-operands and alpha-operators. Order of evaluation of an alpha-expression is always sequential from left to right on an operator-by-operator basis. That is, the result of the first operation is placed in the alpha-receiver and each subsequent operation is conducted left to right on the subsequent new values of the receiver.

For example, in the statement:

```
A$ = B$ AND C$ OR D$
```

The contents of the alpha-operand B$ is first assigned to the alpha-receiver A$. Then the contents of the next alpha-operand A$ is ANDed to the current value of A$. Then the contents of the final alpha-operand D$ is ORed with the current value of A$ and the evaluation is complete. Parentheses cannot be used to alter this order of operations.

LET Alpha Assignment (cont.)

NOTE: The first type of an alpha-expression allows that the initial alpha-operand is optional, and may be omitted. The following is an example of this:

```
A$=AND B$
```

Here, the value of alpha-operand B$ is ANDed with the current value of the alpha-receiver A$ as existed prior to execution of the statement. In effect, this statement is equivalent to A$ = A$ AND B$.

This is not true of the concatenation alpha-expression where the first alpha-operand is mandatory.

If more than one alpha-receiver is specified, assignment is done to the last receiver first. Evaluation proceeds as if a series of LET statements were executed.

Examples:

```
0010 A$=B$ & "ABC" & C$
0010 STR(A$(),10,16)=STR(B$(3),2,4) & STR(B$(4),2,12)
0010 A$,C$(2),D$()=B$ & ALL("X")
0010 A$,STR(B$(4),3,4)=STR(C$,4,6) AND HEX(F0)
0010 Results$=AR$.Custname$ AND ALL ('BitStrip$)

:0005 DIM B$(5)16
:0010 A$="RSTUVWXYZ"
:0020 STR(B$(1),3,4)=A$
:0030 PRINT STR(B$(1),3,4)
:RUN
RSTU

0010 data_rec$.INFO$=STR(C$,1,8)
0010 A$=data_rec$.INFO$
```

Compatibility Issues:

There is no variance in NPL and Wang 2200 Basic-2 for the LET statement itself. However, there are some variances in what reciever and assignments are allowed. Refer to ALL, STR, 'Function-name Numeric Expression, and 'Function-name Literal Expression for more details.

LET Alpha Assignment (cont.)

References:

CONCATENATION
ADD
AND
BOOL
DAC
DSC
ORSUB
XOR
ALL function
BIN function

LET Numeric Assignment

General Form:

```
[LET] numeric-variable [,numeric-variable]... = numeric-expression
```

Where:

```
numeric-expression  =  term [arithmetic-operator term]...

term                =  {numeric-function            }
                       {numeric-scalar-variable     }
                       {numeric-array-element       }
                       {numeric-constant            }
                       {(numeric-expression)        }
                       {-term                       }
                       {user-defined numeric function}
                       {numeric-field expression     }

arithmetic-operator =  {+}
                       {-}
                       {*}
                       {/}
                       {^}

numeric-function     = { INT(n)        } or  { LGT(n)      }
                       { FIX(n)        }     { LOG(n)      }
                       { ABS(n)        }     { EXP(n)      }
                       { SGN(n)        }     { #PI         }
                       { MOD(n,n)      }     { SIN(n)      }
                       { ROUND(n,n)    }     { COS(n)      }
                       { RND(n)        }     { TAN(n)      }
                       { SQR(n)        }     { ARCSIN(n)   }
                       { MAX Function  }     { ARCCOS(n)   }
                       { MIN Function  }     { ATN(n)      }
                       { ERR           }     { LEN Function}
                       { SPACE         }     { NUM Function}
                       { SPACEF}
                       { SPACEK        }     { POS Function}
                       { SPACEP        }     { VAL Function}
                       { SPACEV        }     { VER Function}
                       {SPACEW}
                       { #ID           }
                       { #GOLDKEY      }
                       { #PART         }
                       { #TERM         }
                       { #RECORDLENGTH( record-identifier)   }
                       { #FIELDSTART( field-identifier)      }
                       { #FIELDLENGTH( field-identifier)     }

n                    = numeric-expression
```

LET Numeric Assignment (cont.)

Discussion:

The LET statement is used to assign the result of the evaluation of the expression on the right-hand side of the "=" to the variable or variables on the left-hand side of the "=".

There are two forms of the assignment statement: numeric assignment and alpha assignment. Type conversion is not performed and, therefore, mixed type assignment generates an error. Refer to LET Alpha Assignment statement for details on how to assign values to alpha-receivers.

NOTE: The use of the word LET, if omitted, is assumed.

Evaluation of Numeric-Expressions

As defined by the general form, a numeric-expression may contain a series of terms separated by arithmetic operators. A term may consist of numeric-scalar-variables, numeric-array-elements, numeric-constants, numeric-functions, and numeric-expressions.

Evaluation of numeric-expressions is from left to right, except where affected by the priorities. However, unlike the evaluation of alpha-expressions, the interim results are stored in an internal work area and not placed in any of the receivers until the entire expression is evaluated. Refer to LET Alpha Assignment statement for details on how to assign values to alpha-receivers.

Order of Arithmetic Evaluation

The order of priority for arithmetic-operators is:

1. ^ (exponentiation)
2. - (negation)
3. *, / (multiplication, division)
4. +,- (addition, subtraction)

NOTE: On most keyboards, the exponentiation sign ("^") is the up-arrow key.

The priority of evaluation can be modified by the use of parentheses such that portions of an expression within parentheses are evaluated first.

LET Numeric Assignment (cont.)

For example:

```
A=B+C/D
```

The expression C/D is evaluated first and then added to B.

```
A=(B+C)/D
```

The use of the parentheses changes the order of evaluation so that the expression (B+C) is evaluated first and the result is divided by D.

Nested Expressions

The operands for numeric-functions may themselves be numeric-expressions. In this case, the evaluation of these nested expressions starts with the innermost expression. Where more than one expression exists at the same level of nesting, evaluation proceeds from left to right. The evaluation of each expression follows the rules of arithmetic evaluation stated above. For example, refer to the following:

```
X=INT(MAX(A*4.5,(B-C)/D))
```

This uses five numeric expressions which are evaluated as follows:

1. A*4.5
2. (B-C)
3. Result of (B-C) divided by D
4. MAX of the results of expressions 1 & 3
5. INT of the result of the expression 4 (the MAX Function)

LET Numeric Assignment (cont.)

Examples:

```
0010 X=((A+B)^C/D(1)-4)*INT(C/D(1))
0010 X,Y(1)=MAX(LEN(A$)-1,LEN(STR(B$)))+A+2
0010 X=A+(-B*C/D)
0010 X=AR$.Amount*'Calculate-Discount(Customer_Number)

:0010 LET M=4
:0020 PRINT M
:RUN
4

:0010 W=10: P=15
:0020 J,K,L=W*(P/3)^2
:0030 PRINT J,K,L
:RUN
250         250        250
```

Compatibility Issues:

Due to the use of a different algorithm, results of functions used in a LET statement may differ from functions evaluated on a Wang 2200. In general, however, the functions are accurate to 13 significant digits.

References:

LET Numeric Field Assignment

General Form:

```
[LET]alpha-variable.{field-identifier }[(sub1[,sub2])]=num-exp
                    {<alpha-variable> }
```

Discussion:

Assignment statements permit the left-hand side to be a numeric field reference if the right-hand side is a numeric expression.

Multiple receivers on the left-hand side are not permitted.

Examples:

```
0010 PayrollRecord$.Federal_Withholding=Gross*Rate
0010 InputScreenHeader$.Cursor_Position_Row=0
0010 Employee_Record$.<Deduction_Name$>=0
0010 Employee_Record$.Miscellaneous_Deductions(I)=Transaction_Misc(I)
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

RECORD
FIELD

LGT Function

General Form:

```
    LGT (numeric-expression)
```

Discussion:

The LGT function is used to derive the common base 10 logarithm of a numeric-expression greater than 0. This is valid wherever a numeric-expression is legal.

Examples:

```
:0010 B3=500
:0020 A=LGT(B3+9)
:0030 PRINT A
:RUN
2.7067177823368

:0010 C=1000+LGT(2000)
:0020 PRINT C
:RUN
1003.30102999566
```

Compatibility Issues:

Due to the use of different algorithms, results of this function may differ from the Wang 2200. In general, however, the function is accurate to 13 significant digits.

References:

LIMITS

General Form:

Form 1:

```
LIMITS T [device-address,] file-name, start, end
         [file-number,   ]
         [<address-var>, ]
                                          [ ,[used] [,status]]
```

Form 2:

```
LIMITS T [file-number,] start, end, current
```

Where:

```
start   = numeric-receiver which will receive the starting sector
          address.

end     = numeric-receiver which will receive the ending sector
          address.

used    = numeric-receiver which will receive the number of sec-
          tors used.

status  = numeric-receiver which will receive the current status
          of the file.

current = numeric-receiver which will receive the current sector
          address.
```

NOTE: The use of this statement is not recommended. Refer to Niakwa Data Manager as a better alternative.

LIMITS (cont.)

Discussion:

The LIMITS statement is used to obtain sector address information about a cataloged file. There are two forms of the LIMITS statement. Using Form 1, the file-name is specified and access is made to the catalog to determine the requested information. Using Form 2, the file-name is not specified and information is retrieved from the specified device slot in the Internal Device Table.

Form 1

Form 1 of the LIMITS statement is used to determine the starting sector, ending sector, number of sectors used, and the status for the specified file-name. The used parameter, if specified, is obtained from the trailer sector of the specified file. Refer to Section 7.3.7 of the Programmer's Guide for details on the trailer sector. If a variable is not specified for the status and the file-name does not exist, an error occurs when the LIMITS statement is executed.

The status codes are:

-2	Scratched data file
-1	Scratched program file
0	File not found
1	Program file
2	Data file

Form 2

Form 2 of the LIMITS statement looks for the specified file-number in the Internal Device Table, returning the starting, ending, and current sector addresses for that file-number. If file-number is not specified, the default slot (#0) is used. If the file number has not been opened (using DATA LOAD DC OPEN or DATA SAVE DC OPEN), zeros are returned for all values.

LIMITS (cont.)

Examples:

```
0010 LIMITS T/D10,"EMPLOYEE",A,B
0010 LIMITS T#X,A$,Q1,Q2,Q3
0010 LIMITS T#6,STR(N$,5,8),S,E,U,R
0010 LIMITS T/D30,X$(3,4),R1,R2,R3,R4
0010 LIMITS T#2,"PAYROLL",P,P(1),P(2),P(3)
0010 LIMITS T#8,S,E,C
0010 LIMITS T<A$>,S,E,C
0010 LIMITS T A,B,C
```

Compatibility Issues:

The LIMITS statement has been extended in NPL as follows:

1. The diskimage address for Form 1 may be an arbitrary /xxx address (the address need not be established in an internal device table slot using a previous SELECT statement).

2. The "used" variable need not be specified. If the "used" parameter is not requested, the LIMITS statement executes faster, since substantially less head movement of the disk is required.

3. If the "used" variable is the same as the "type" variable, the statement is executed as though the "used" variable was not specified.

4. Previous versions of NPL allowed the syntax:

   ```
   LIMITS T [device-address,]file-number,start,end, [used][,status]
            [file-number,   ]
   ```

   The trailing comma is no longer supported.

5. Use of the address-var parameter is supported only on NPL Revision 3.0 or greater and is not supported on the Wang 2200.

References:

DATA LOAD DC OPEN
DATA SAVE DC OPEN
Internal Device Table - Section 7.2.3 of the Programmer's Guide
Disk Devices - Section 7.2 of the Programmer's Guide

LIMITS INDEX

General Form:

```
     LIMITS INDEX T [file-number,  ] num-var1, num-var2, num-var3,
                    [disk-address, ]                     num-var4
                    [<address-var>,]
```

Where:

```
     num-var1 = a numeric-variable that will receive the number of in-
                dex sectors of the specified diskimage.

     num-var2 = a numeric-variable that will receive the value of the
                CURRENT END of the specified diskimage +1.

     num-var3 = a numeric-variable that will receive the value of the
                END CATALOG of the specified diskimage +1.

     num-var4 = a numeric-variable that will receive the hash type of
                the specified diskimage.  A value of 0 indicates nor-
                mal hash type, a value of 1 indicates alternate hash
                type as created by SCRATCH DISK '.
```

NOTE: The use of this statement is not recommended. Refer to Niakwa Data Manager as a better alternative.

Discussion:

The LIMITS INDEX statement reads the index sector of the specified diskimage and returns the number of index sectors, current end (+ 1), end catalog (+ 1), and hash type values into the specified numeric-variables. This permits program inspection of these values without performing direct access to sector zero and converting the binary values.

LIMITS INDEX respects the EXT clause of the device equivalence definition for the diskimage. That is, if EXT=Y is not specified, LIMITS INDEX disregards values stored in bytes 7 and 8 of sector zero when calculating the values to return for CURRENT END and END CATALOG. Refer to Section 7.3.10 of the Programmer's Guide for further details on extended diskimages. Refer to Section 7.3.6 of the Programmer's Guide for further details on the internal structure of sector zero of a diskimage file.

LIMITS INDEX (cont.)

Typical use of LIMITS INDEX is for determination of the amount of space available in a diskimage or in determination of the END CATALOG value to specify in a MOVE END operation or the LS and END CATALOG value to specify in a MOVE operation.

For example:

```
0010 X=2000                  :REM I need a 2000 sector file
0020 LIMITS INDEX T#1,A,B,C,D
0030 IF C-B<X THEN 100       :REM Not enough space - try MOVE END
0040 DATA SAVE DC OPEN T#1,(X)"MYFILE"<
               :ERROR GOTO 130
0050 STOP "Successful completion"
0100 MOVE END T#1,=B+X<
               :ERROR GOTO 120
0110 GOTO 40                   :REM Now I have enough room
0120 REM Can't do it at all - advise operator
0130 REM Error on DATA SAVE DC OPEN - advise operator
```

NOTE: LIMITS INDEX assumes that sector zero of the specified diskimage file contains valid information. No attempt is made to validate the information returned.

Examples:

```
0010 LIMITS INDEX T/D10,A,B,C,D
0010 LIMITS INDEX T#1,A,X(1),X(2),X(3)
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

LIMITS INDEX is not supported on the Wang 2200.

References:

LIST DC
MOVE
MOVE END
$DEVICE
NPL Diskimages - Section 7.3.6 of the Programmer's Guide
Extended Diskimages - Section 7.3.10 of the Programmer's Guide

LINPUT

General Form:

```
LINPUT [literal,] [?] [-]alpha-variable
```

Where:

```
literal = optional text to be displayed on the screen.

-       = causes contents of alpha-variable to appear underlined
          on the screen.

?       = causes LINPUT to begin in DEFFN Mode.
```

Discussion:

LINPUT is used to perform field type entry of an alpha-variable.

Upon execution of the LINPUT statement, the current value of the specified alpha-variable is displayed as a field of a length equal to the length of the alpha-variable with the cursor appearing at the beginning of the value. If the optional "-" is specified, the contents of the alpha-variable is displayed underlined. At this point, the operator may enter a new value or EDIT the existing value.

The specified variable in a LINPUT statement serves in place of the EDIT buffer. That is, all data entry/edit is performed directly on the contents of the alpha-variable.

Pressing the RETURN key terminates the LINPUT operation with the current value of the field on the screen being stored in the specified alpha-variable.

EDIT Capabilities:

If the "?" parameter is specified, the LINPUT begins in DEFFN Mode as opposed to Edit mode.

LINPUT (cont.)

In DEFFN Mode:

- Program assigned Special Function keys are enabled.
- The cursor appears steady.
- Keypad keys (EAST, WEST, NORTH, SOUTH, INSERT, DELETE) may be used to manipulate data in the field.

In Edit Mode:

- Program assigned Special Function keys are disabled (except SF'126 and SF'127).
- All Edit mode Special Function keys are enabled and may be used to manipulate data in the field (except for LINE INSERT/LINE DELETE).
- The cursor appears blinking.
- Keypad keys (EAST, WEST, NORTH, SOUTH, INSERT, DELETE) may be used to manipulate data in the field.
- If all data in the field is erased by use of the LINE ERASE key, the initial value may be recalled by use of the RECALL key.

Refer to Chapter 5 of the Programmer's Guide for details on EDIT mode capabilities.

In both Edit and DEFFN Mode, cursor movement is limited to the specified length of the alpha-variable.

Regardless of the initial mode (DEFFN versus Edit), the operator may change modes at any time during execution of the statement by pressing the EDIT key.

Use of Special Function Keys to Access DEFFN's:

Special Function keys may be used to access text definition subroutines or executable subroutines when in DEFFN Mode.

LINPUT (cont.)

If a text definition subroutine is called, the specified text is placed in the alpha-receiver starting at the current cursor position, the cursor is located at the end of the added text, and execution of the LINPUT statement continues. If a HEX(0D) is present in the text, the LINPUT statement is completed as if the RETURN key were pressed.

If an executable subroutine is called, execution of the LINPUT is terminated as if RETURN was pressed, then the subroutine is called. Upon RETURN from the subroutine, the statement following the LINPUT is executed.

All the functions of the Line Editor are available to allow entry or correction of the LINPUT line. Refer to Section 5.4 of the Programmer's Guide for details.

In addition, the initial mode of the Line Editor for a LINPUT statement may be set so that entered data overstrikes or is automatically inserted into the entered line. Also, the operation of the INSERT key for a LINPUT statement can be set so that it either inserts a single space or switches between insert and overstrike modes. These choices for these options are set using byte 44 of the $OPTIONS system variable. Refer to $OPTIONS for more details.

NOTE: **Attempting to call subroutines which contain parameters is not permitted (an alarm is sounded and the key is ignored).**

Examples:

```
0010 LINPUT "ENTER THE DISK ADDRESS",B$
0010 LINPUT STR(D$(),1,20)
0010 LINPUT "IS ANSWER CORRECT? ",Q9$
0010 LINPUT -A$
0010 LINPUT "Edit Programmer Name"?-Z$
```

Compatibility Issues:

In Wang 2200 Basic-2, the maximum length of an alpha-variable used for LINPUT operations is 480 bytes. In NPL, the maximum length is 512 bytes.

Execution of marked subroutines with parameters during response to LINPUT is supported by Wang 2200 Basic-2. NPL does not support this feature.

LINPUT (cont.)

References:

INPUT
KEYIN
DEFFN’
DEFFN’
Edit Mode versus DEFFN Mode - Section 5.4 of the Programmer’s Guide

LIST (General Parameters)

General Form:

```
LIST [title] [S]
             [F]
```

Where:

```
title = an optional descriptive title; may be a literal-string or
        alpha-variable.

S     = specifies that a page break be performed.

F     = specifies that no page break be performed.
```

Discussion:

This section discusses topics that are common to all forms of the LIST statement. The documentation for each individual LIST statement refers to this section.

General Parameters for all LIST Statements

LIST output that is directed to the screen is displayed one screen at a time. That is, a page break is issued to prevent LIST output from rapidly scrolling up the screen. When a page break is issued, the system halts the list operation and display the message "--MORE--" in the lower right corner of the screen. Pressing the RETURN key at this prompt continues the LIST operation. If the page break should occur in the middle of a multi-statement line, the multi-statement line is broken up at the correct line with the remainder of the line being displayed after continuing from the page break. The number of lines between page breaks can be controlled using the current SELECT LINE entry in the Internal Device Table. This page break can be suppressed by specifying the F parameter.

LIST output that is directed to the printer is printed continuously. That is, the system does not halt between pages. If the S parameter is not specified, output is printed continuously from one page to the next. If the "S" parameter is specified, a form-feed character is automatically inserted into the output stream based on the SELECT LISTLINE value (default is 55 lines).

LIST (General Parameters) (cont.)

When an optional title is specified, the title is listed with two blank lines appearing at the beginning of the list. If the output is directed to the screen, the title is highlighted (printed brighter than normal). If output is directed to a printer and the "S" parameter is specified, the title prints on every page of the listing.

NOTE: When LIST is executed as a program statement on listings directed to the screen, the "S" parameter is ignored.

Compatibility Issues:

Generation of page breaks when listing to a printer is supported only in NPL revision 3.0 or greater.

This statement is supported only with Release 3.0 or greater.

References:

LIST

General Form:

```
LIST [title] [S] [D] [([low-line][,[high-line]])]
             [F]          [line-number1][,[line-number2]]
```

Where:

```
title        =  an optional descriptive title; may be a literal-
                string or alpha-variable.

S             = specifies that a page break be performed.

F             = specifies that no page break be performed.

D             = specifies de-compressed output.

low-line      = lowest line-number for which to show references.

high-line     = highest line-number for which to show references.

line-number1 =  lowest line-number to list.

line-number2 =  highest line-number to list.
```

Discussion:

Refer to LIST general parameters section for details on parameters common to all LIST statements.

The LIST command produces a listing of the program currently in memory.

LIST (cont.)

The LIST function refers to program text in the current list module. This is set to the currently executing module whenever a program HALTs or continues, or when changed using the MODULE command, and can be referenced using "LIST DT".

The optional "D" parameter causes the LIST output to be displayed in decompressed form (multi-statement lines are displayed 1 statement per line). In addition to decompressed form, the "D" parameter also displays line-numbers that are referenced at other points in the program with a "-" designation in front of the line-number. The "D" parameter also affects the listing of special remark statements:

- REM% statements are displayed with a blank line before and after the remark text. If the list output is displayed to the screen, the remark text is also highlighted (printed brighter than normal).

- REM%^ statements are displayed with a blank line before and after the remark text. If the list output is directed to the printer, a page break is issued before the text is printed.

The optional line-number range parameters operate as follows:

- If only line-number1 is specified, only that specific program line is listed.

- If line-number1, (comma) is specified, all program lines starting at line-number1 are listed in ascending sequence.

- If ,(comma)line-number2 is specified, all program lines starting at the lowest ASCII sequence up to and including line-number2 are listed.

- If line-number1,line-number2 is specified, all program lines within the range of line-number1 to line-number2 inclusive are listed.

- If no line-numbers are specified, the entire program in memory is listed.

The optional low/high range parameters are used to specify the range of lines accessed for determining whether or not to place a "-" before a line number listed when the "D" parameter is used. The "-" indicates that the line-number is referenced at other points in the program within the specified low/high range. These operate as follows:

LIST (cont.)

- If only low-line is specified, only that specific program line is accessed.
- If low-line, (comma) is specified, all program lines starting at low-line are accessed in ascending sequence.
- If ,(comma)high-line is specified, all program lines starting at the lowest ASCII sequence up to and including high-line are accessed.
- If low-line,high-line is specified, all program lines within the range of low-line to high-line inclusive are accessed.
- If no line-numbers are specified, the entire program in memory is accessed.

LIST performs no operation on the non-interpretive form of the RunTime Program.

Examples:

```
:LIST
:LISTD
:LISTD(0,100)
:LIST 2000,3000
:LISTD(0,100)2000,3000

:LIST
0010 FOR I=1 TO 10: PRINT I: NEXT I
0020 PRINT "THIS IS A TEST": J$="XYZ"
0030 A=1: B=2: C=3: PRINT A,B,C,J$
0040 GOSUB 200
0050 IF Q(1)=1 THEN 20
1000 DEFFN’15 "LISTDT";HEX(0D)
1010 DEFFN’16 "LISTSD";HEX(0D)
```

LIST (cont.)

```
:LISTD
 0010 FOR I=1 TO 10
    : PRINT I
    : NEXT I
 0020 PRINT "THIS IS A TEST"
        : J$="XYZ"
 0030 A=1
    : B=2
    : C=3
    : PRINT A,B,C,J$
 0040 GOSUB 200
 0050 IF Q(1)=1 THEN 20
 1000 DEFFN'15 "LISTDT";HEX(0D)
 1010 DEFFN'16 "LISTSD";HEX(0D)

:LIST"Lines 20 through 30"D20,30

Lines 20 through 30
 0020 PRINT "THIS IS A TEST"
    : J$="XYZ"
 0030 A=1
    : B=2
    : C=3
    : PRINT A,B,C,J$

A$=Lines 20 through 30"
:LIST A$ D(20,30)20,30
Lines 20 through 30
 0020 PRINT "THIS IS A TEST"
    : J$="XYZ"
 0030 A=1
    : B=2
    : C=3

:PRINT A,B,C,J$
```

NOTE: The "D" parameter causes the output to be displayed in decompressed format with line-numbers referenced within the low-line,high-line range displayed with the "-" designation.

LIST (cont.)

```
:0010 GOTO 20: GOTO 30
:0015 GOTO 25
:0020 GOTO 30
:0025 GOTO 10
:0030 GOTO 20
:LISTD

-0010 GOTO 20

    : GOTO 30

 0015 GOTO 25

-0020 GOTO 30

-0025 GOTO 10

-0030 GOTO 20

:LISTD(10,10)

0010 GOTO 20

    : GOTO 30

0015 GOTO 25

-0020 GOTO 30

 0025 GOTO 10

-0030 GOTO 20
```

NOTE: With the low-line,high-line range of 10,10, LISTD places a "-" only at lines 20 and 30 which are the only lines referenced in the range 10,10.

Compatibility Issues:

LIST is supported on NPL Revisions 2.00 and greater.

The "F" parameter is not supported in Wang 2200 Basic-2.

Low-line,high-line ranges are supported only on NPL Revision 3.0 or greater and are not supported on the Wang 2200.

References:

Inspection of Program Text - Section 6.5 of the Programmer’s Guide

LIST

General Form:

```
LIST [title] [S] #[*] [([low-line  ][,[high-line]]) ]
             [F]
                      [line-number1][,[line-number2]]
```

Where:

```
title        = an optional descriptive title; must be a literal-
               string.

S            = specifies that a page break be performed.

F            = specifies that no page break be performed.

*            = list program line as opposed to just line-number.

line-number1 = low range of line-numbers to be cross-referenced.

line-number2 = high range of line-numbers to be cross-referenced

low-line     = lowest line-number for which to show references.

high-line    = highest line-number for which to show references.
```

Discussion:

The LIST # command produces a cross-reference listing of all references to specific line-numbers within the specified range of lines of the program in memory.

The LIST function refers to program text in the current list module. This is set to the currently executing module whenever a program HALTs or continues, or when changed using the MODULE command, and can be referenced using LIST DT.

Refer to LIST general parameters for details on general parameters for all LIST statements.

The optional line-number range parameters operate as follows:

LIST # (cont.)

- If only line-number1 is specified, a cross-reference is performed only on that specific line-number.
- If line-number1, (comma) is specified, all line-numbers starting at line-number1 are cross-referenced in ascending ASCII sequence.
- If ,(comma)line-number2 is specified, all line-numbers starting at the lowest ASCII sequence up to and including line-number2 are cross-referenced.
- If line-number1,line-number2 is specified, all line-numbers within the range of line-number1 to line-number2 inclusive are cross-referenced.
- If no line-numbers are specified, a cross-reference is performed on all line-number references.

The optional low/high range parameters are used to specify the range of lines accessed from which references are displayed. These operate as follows:

- If only low-line is specified, only that specific program line is accessed.
- If low-line, (comma) is specified, all program lines starting at low-line are accessed in ascending sequence.
- If ,(comma)high-line is specified, all program lines starting at the lowest ASCII sequence up to and including high-line are accessed.
- If low-line,high-line is specified, all program lines within the range of low-line to high-line inclusive are accessed.
- If no line-numbers are specified, all program text for the current list module is accessed.

If a line-number is referenced by 1 or more statements within the low/high range, but is not present in the program range being listed, the line-number are listed in the cross-reference with a "?" in front of the line-number.

LIST # (cont.)

The default format for the LIST # command lists only the cross-referenced line-numbers. Specifying the "*" parameter displays the statement on the line where the line number was referenced. In addition, a number of colons (":") precede the statement to indicate how many statements precede the referenced statement.

NOTE: This clause will change for each LIST statement noted.

LIST # performs no operation on the non-interpretive form of the RunTime program.

Examples:

```
:LIST #
:LIST # 60,
:LIST "Referenced Program lines through line 100"# * ,100
:LIST # 50,150
:LIST # * 1000,
:LIST # (0,2000)50,150
:LIST # (8000,)
:LIST # *(100,),200
2000 REM
   : REM     SAMPLE PROGRAM
   : REM
2010 GOSUB '100  : REM open data file
2020 GOSUB '101    : REM read a record
2030 IF END THEN 2100             : REM quit if end of file
2040    IF F$="X" THEN R1=R1+1
   :              ELSE R2=R2+1    : REM update R1 or R2 record
                                        count
2050 A=MIN(A,F2,F9*2)             : REM compute min of fields F2,F9
2060 B=MAX(B,F3,F8)               : REM compute max of F3 and F8
2070 F2,F3,F8,F9=0 : REM reset data values
2080 GOSUB '102    : REM update data record
2090 GOTO 2020     : REM iterate until eof
2100 GOSUB '103    : REM close file
2105 PRINT "MIN OF F2,F9 IS",A    : REM display results
2110 PRINT "MAX OF F3,F8 IS",B
2120 STOP
2130 DEFFN'100 : RETURN           : REM This subr opens a file
2140 DEFFN'101 : RETURN           : REM this subr reads a record
2150 DEFFN'102 : RETURN           : REM this subr writes a record
2160 DEFFN'103 : RETURN           : REM this subr closes a file
:LIST #
 2020 - 2090
 2100 - 2030

:LIST # *
2020
2090    GOTO 2020
2100
2030 IF END THEN 2100

:LIST #(0,2050)
2020 - 2090
```

LIST # (cont.)

```
:LIST
:0010 GOTO 20: GOTO 30
:0020 GOTO 30
:0025 GOTO 10
:0030 GOTO 20

:LIST #
0010 - 0025
0020 - 0010 0030
0030 - 0010 0020

:LIST#(20,30)

 0010 - 0025
 0020 - 0030
 0030 - 0020

:LIST#(20,30)20,30

 0020 - 0030
 0030 - 0020
```

Compatibility Issues:

The "*" parameter is not valid in Wang 2200 Basic-2.

LIST # is supported on NPL Revisions 2.00 and greater.

In Wang 2200 Basic-2, if a line-number is referenced more than once from a program line (multi-statement line), only 1 reference for the program line appears in the LIST # output. In NPL, a reference is made for each reference in the program line.

Low-line,high-line ranges are supported only on NPL Revision 3.0 or greater and are not supported on the Wang 2200.

Prior to NPL Release IV, the "*" option would display all statements on the line containing the reference.

References:

LIST DT
MODULE
Inspection of Program Text - Section 6.5 of the Programmer's Guide

LIST ’

General Form:

```
LIST [title] [S] ’[*] [([low-line][,[high-line]])     ]
             [F]
                      [deffn-1][,[deffn-2]            ]
```

Where:

```
title     = an optional descriptive title; must be a literal-
            string.

S         = specifies that a page break be performed.

F         = specifies that no page break be performed.

*         = list program line as opposed to just line-number.

deffn-1   = low range of subroutines to be displayed.

deffn-2   = high range of subroutines to be displayed.

low-line  = lowest line-number for which to show references.

high-line = highest line-number for which to show references.
```

Discussion:

The LIST’ command produces a cross-reference listing of all references in a program to DEFFN’ subroutines using GOSUB’ statements.

The LIST function refers to program text in the current list module. This is set to the currently executing module whenever a program HALTs or continues, or when changed using the MODULE command, and can be referenced using LIST DT.

Refer to LIST Statement for details on general parameters for all LIST statements.

LIST’ statements are extended to include named DEFFN routines.

LIST ’ (cont.)

NOTE: In ranges of DEFFN’ names, numbered DEFFIN’s sort numerically but named DEFFIN’s sort lexicographically (all numbers appear before any names).

For example:

’2	**appears before ’12**	**<- numerical**
’9999	**appears before ’65535**	
’Aardvark	**appears before ’Zebra**	
’f10000	**appears before ’f9**	**<-lexical**

A LIST’ range that ends at 65535 is equivalent to ’all ranges after start value’. It is not possible to specify a range that ends exactly at 65535.

The optional range parameters (deffn-1,deffn-2) operate as follows:

- If only deffn-1 is specified, a cross-reference is performed only on that specific marked subroutine.
- If deffn-1, (comma) is specified, all marked subroutines starting at deffn-1 are cross-referenced in ascending ASCII sequence.
- If ,(comma)deffn-2 is specified, all marked subroutines, starting at the lowest ASCII sequence up to and including deffn-2, are cross-referenced.
- If deffn-1,deffn-2 is specified, all marked subroutines within the range of deffn-1 to deffn-2, inclusive, are cross-referenced.
- If no deffn range parameters are specified, a cross-reference is performed on all marked subroutines.

The optional low/high range parameters are used to specify the range of lines accessed from which references (GOSUB ’) and definitions (DEFFN ’) are displayed. These operate as follows:

- If only low-line is specified, only that specific program line is accessed.
- If low-line, (comma) is specified, all program lines starting at low-line are accessed in ascending sequence.

LIST ’ (cont.)

- If ,(comma)high-line is specified, all program lines starting at the lowest ASCII sequence, up to and including high-line, are accessed.
- If low-line,high-line is specified, all program lines within the range of low-line to high-line, inclusive, are accessed.
- If no line-numbers are specified, the entire program in the current list module is accessed.

In addition to listing subroutines defined by DEFFN ’, LIST ’ also lists subroutines defined by external subroutines. DEFFN ’s defined in external subroutines are always displayed by LIST ’, regardless of any low-line, high-line range specified.

NOTE: Unless the rtpdef_next_number field is defined in the external routines, LIST ’ may respond slowly the first time executed when external subroutines are loaded. Refer to the NPL External Subroutine Development Kit documentation for further details on external subroutines.

If a DEFFN’ subroutine is called by one or more GOSUB’ statements but is not defined in the low-line, high-line range being accessed, the subroutine is cross-referenced with the line-number displayed as ("????").

LIST ’ performs no operation on the non-interpretive form of the RunTime program.

Specifying the "*" parameter displays the statement on the line where the DEEFN’ subroutine was referenced. In addition, a number of colons ":" precede the statement to indicate how many statements precede the referenced statement.

Examples:

```
:LIST’
:LIST’10,31
:LIST"SUBROUTINES"’ *
:LIST"Subroutine ’31"’ 31
:LIST ’(100,200)
:LIST ’(2000,)10,31
:LIST ’ *(,4000)24,

2000 REM
   : REM     SAMPLE PROGRAM
   : REM
2010 GOSUB ’fileOpen              : REM open data file
2020 GOSUB ’101    : REM read a record
2030 IF END THEN 2100             : REM quit if end of file
```

LIST ' (cont.)

```
2040    IF F$="X" THEN R1=R1+1
   :              ELSE R2=R2+1: REM update R1 or R1 rcd
.                                          counter
2050    A=MIN(A,F2,F9*2)          : REM compute min of fields F2,F9
2060    B=MAX(B,F3,F8)            : REM compute max of F3 and F8
2070    F2,F3,F8,F9=0             : REM reset data values
2080    GOSUB '102 : REM update data record
2090    GOTO 2020  : REM iterate until eof
2100 GOSUB '103    : REM close file
2105 PRINT "MIN OF F2,F9 IS",A    : REM display results
2110 PRINT "MAX OF F3,F8 IS",B
2120 STOP
2130 DEFFN'fileOpen : RETURN      : REM This subr opens a file
2140 DEFFN'101 : RETURN           : REM this subr reads a record
2150 DEFFN'102 : RETURN           : REM this subr writes a record
2160 DEFFN'103 : RETURN           : REM this subr closes a file

:LIST '
2130 DEFFN'fileOpen
      - 2010
2140 DEFFN'101
     - 2020
2150 DEFFN'102
     - 2080
2160 DEFFN'103
     - 2100
:LIST '(2100,)

2130 DEFFN'fileOpen
2140 DEFFN'101
2150 DEFFN'102
2160 DEFFN'103
     - 2100

:LIST '(2100,)102,
2150 DEFFN'102
2160 DEFFN'103
     - 2100
```

Compatibility Issues:

The "*" parameter is not valid in Wang 2200 Basic-2.

LIST ' is supported on NPL Revisions 2.00 and greater.

Wang 2200 Basic-2 does not support the use of range parameters with LIST ' output.

Low-line,high-line ranges are supported only on NPL Revision 3.0 or greater and are not supported on the Wang 2200.

Prior to NPL Release IV, the "*" option would display all statements on the line containing the reference.

LIST ' (cont.)

References:
DEFFN'
GOSUB'
LIST DT
Inspection of Program Text - Section 6.5 of the Programmer's Guide

LIST DC

General Form:

```
LIST[title] [S] DC T[device-address,][restrict[,restrict]...][W]
            [F]     [file-number,  ]
                    [<address-var>,]
```

Where:

```
title      = an optional descriptive title; must be a literal-
             string.

S          = specifies that a page break be performed.

F          = specifies that no page break be performed.

restrict   = {[FILE  rel-op] alpha-mask          }
             { TYPE  rel-op  alpha-mask          }
             { START rel-op  numeric-expression  }
             { END   rel-op  numeric-expression  }
             { USED  rel-op  numeric-expression  }
             { FREE  rel-op  numeric-expression  }
             { DATE  rel-op  alpha-mask          }
             { TIME  rel-op  alpha-mask          }

W          = Specifies that output from LIST DC should consist
             only of the names of the files selected displayed
             across the screen.

rel-op     = relational operator {<,=,>,<=,>=,<>}.

alpha-mask = alpha-variable or alpha-literal.
```

Discussion:

The LIST DC format of the LIST command produces a listing of files in the diskimage file specified by the device-address.

LIST DC (cont.)

Any diskimage file currently defined in the device equivalence table can be specified in the LIST DC command using the device address parameter. If no device address is specified, the diskimage address currently defined as the default diskimage (#0) in the Internal Device Table is assumed.

Refer to LIST general parameters section for details on parameters common to all LIST statements.

The listing is comprised of:

General information about the diskimage:

- Device Equivalence
- Number of Index Sectors
- End Catalog
- Current End

Information about each specified cataloged file on the disk (refer below for specification methods):

File name	
File Type	Scratched or Not, Program or Data
Start Sector	(Beginning sector address of the file)
End Sector	(Ending sector address of the file)
Sectors Used	(Number of sectors occupied by file)
Sectors Free	(Number of unused sectors in the file)
Date Stamp	(Date the file was last modified)
Time Stamp	(Time the file was last modified)

LIST DC (cont.)

Refer to Section 7.3.6 of the Programmer's Guide, (Internal Structure of Diskimages) for additional details about these parameters.

Specifying Files

The LIST DC command allows restriction of the file listing by specification of key words related to information about the file, followed by a relational operator, followed by a mask. As the catalog index is read, file parameters are matched against the specified mask as required by the relational operator. Only files meeting the specified requirements are listed. Multiple restrictions may be specified, in which case only files meeting all requirements are listed.

For keywords which represent alpha data, the mask must be a literal or alpha-variable. For keywords which represent numeric data the mask must be a valid numeric expression of which the integer portion is used.

For alpha masks, standard wildcard usage is supported. That is, a "?" in any position matches any character in that position. An "*" indicates that any characters from the position of the asterisk to the end of the field match.

The key words available for file specification are:

FILE	Eight-byte alpha
TYPE	Two-byte alpha. Byte 1 is "S" if the file is scratched; blank if not scratched. Byte 2 is "P" for program files; "D" for data files.
START	Numeric
END	Numeric
USED	Numeric
FREE	Numeric
DATE	Eight-byte alpha in the format yy/mm/dd
TIME	Eight-byte alpha in the format hh:mm:ss

LIST DC (cont.)

NOTE: If no keyword is specified, the key word FILE and the relational operator "=" are assumed.

LIST DC performs no operation on the non-interpretive form of the RunTime program.

As of Revision 3.0 of NPL, the file name, file type, and file status (scratched or not scratched) are stored in the file trailer sector as well as the file index. LISTDCT checks this information and displays a "?" at the end of each file name line where the information in the trailer sector does not exist or does not match the index. This "?" is not displayed if the "W" option is specified.

NOTE: File trailer information can be established by use of a MOVE (form 1) statement.

Examples:

```
:LIST DC T/D32,
$DEVICE(/D32) ="/BASIC2C/PROGS.BS2"
INDEX SECTORS =     10
END SECTORS   =    265
CURRENT END   =    265

FILE    TYPE START   END  USED  FREE    DATE     TIME
2CCOPY    P     89   176    62    26  86/06/01 04:10:31
2CRCVR    P    177   265    72    17  86/05/30 08:59:01 ?
2CBCKP    P     10    88    62    17  86/06/04 15:21:54
```

NOTE: The question mark at the end of the file name line, if present, indicates that filename, type, and status contained in the trailer does not match that contained in the index.

```
:LIST DC T/D32,W
$DEVICE(/D32) ="/BASIC2C/PROGS.BS2"
INDEX SECTORS =     10
END SECTORS   =    265
CURRENT END   =    265

2CCOPY  2CRCVR  2CBCKP

:SELECT #1 D32                    would produce the same listing as
:LIST DCT#1                       LIST DCT/D32 in previous example.

:A$="D32"                         would produce the same listing as
:LIST DCT<A$>                     LIST DCT/D32 in previous example.

:LIST DCT "AR*"                   lists all files beginning with AR
                                  (default key word and relational operator
                                  if none specified is "FILE =").
```

LIST DC (cont.)

```
:LIST DCT "AR*",DATE>="86/01/01"  Lists all files with a name
                                  starting with "AR" and a date
                                  stamp of January 1, 1986 or later.

:LIST DCT TYPE = "?P"             List all programs, scratched and active.

:LIST DCT TYPE = "SP"             List all scratched programs.

:LIST DCT TYPE = " P", DATE="86/09/30", START>=5000

                                  List all active (non-scratched) programs
                                  with a date stamp of Sept. 30, 1986 which
                                  start at sector 5000 or higher on the diskimage.

:LIST DCT "?? '*"                 Lists all files with a space in 3rd
                                  position, and an apostrophe in the 4th
                                  position (default condition if none
                                  specified is "FILE =").

:LIST DCT TYPE="S?"               Lists all scratched files.

:LIST DCT FREE >0                 Lists files with non-zero free space.

:LIST DCT START>=3000             Lists all files which start at or
                                  after sector 3000.
```

Compatibility Issues:

Wang Basic-2 supports a method of file selection similar to NPL. However, the Wang syntax is different than NPL and is not fully supported. In addition, the Wang Basic-2 implementation is limited to the use of a filename mask and specification of a specific file type.

LIST DC is supported on NPL Revision 2.00 and greater.

The optional W parameter is supported on NPL Revision 3.0 or greater.

Use of the address-var parameter is supported only on NPL Revision 3.0 or greater and is not supported on the Wang 2200.

References:

Inspection/Modification of Environment - Section 6.6 of the Programmer's Guide

LIST DIM

General Form:

```
LIST [title] [S] DIM [*] [var1][,[var2]]
             [F]
```

Where:

```
title = an optional descriptive title; must be a literal-string.

S     = specifies that a page break be performed.

F     = specifies that no page break be performed.

*     = causes the contents of variables in the specified range
        to be displayed.

var1  = low variable in range to be displayed.

var2  = high variable in range to be displayed.
```

Discussion:

The LIST DIM command displays the list of variables currently defined in memory within the specified range of variables in alphabetical order.

Variables displayed are those which can be referenced from the current context (executing module and function).

LIST DIM (cont.)

The information displayed for each variable is:

- An indicator of each variable’s allocation status as:

DIM	Module private non-common static variables
COM	Module private common static variables
DIM /RECURSIVE	Function private recursive variables and by-value parameters
DIM /STATIC	Function private static variables
DIM /PUBLIC	Public variables
/POINTER	Function private by-reference parameters

- If the variable is a FIELD or RECORD identifier, a keyword indicating this is displayed

- The variable name

- Current array dimension (if array-variable)

- Element length (if alpha-variable)

- For Function private by-reference (/POINTER) parameters, the name of the variable to which parameter references is displayed, if this can be determined.

- Optionally, by specifying the [*] parameter, the element value is displayed: if a numeric-variable, the numeric value is displayed; if an alpha-variable, the string value is displayed in both ASCII (in quotes) and HEX() representation. If a string value is longer than 16 bytes, the value is displayed on multiple lines, with the starting STR() index of each part at the beginning of each line. Non-displayable HEX codes which do not have printable character representations are displayed in string value as ".". If the variable is a FIELD identifier, the values of the #FIELDSTART, #FIELDLENGTH and $FIELDFORMAT functions are displayed. If the variable is a RECORD identifier, the value of the #RECORDLENGTH function is displayed.

Refer to LIST General Parameters section for details on parameters common to all LIST statements.

LIST DIM (cont.)

The optional variable range parameters operate as follows:

- If only var1 is specified, LIST DIM output for only that specific variable is generated.
- If var1, (comma) is specified, LIST DIM output for variables starting with var1 in ascending ASCII sequence is generated.
- If ,(comma)var2 is specified, LIST DIM output for variables starting at the lowest ASCII sequence up to and including var2 is generated.
- If var1,var2 is specified, LIST DIM output for variables within the range of var1 to var2 inclusive is generated.
- If no variable range parameters are specified, LIST DIM output is generated for all variables currently defined in memory.

If exactly 1 type of variable (numeric-scalar, alpha-scalar, numeric-array, alpha-array) is specified in var1 and var2, only variables of that type are listed.

Array variables are specified in a LIST DIM statement using a special syntax. The array designator is specified followed by an open parenthesis "(". For example, the arrays S$() and N() would be specified by:

```
0010 LIST DIM S$(, N(
```

LIST DIM displays all variables defined in memory, even if not referenced in the current program text. However, if the program has not yet been resolved, variables referenced in the program text may not be defined yet. Refer to LIST V for a cross-reference of variables referenced in the program text.

LIST DIM performs no operation when executed by the non-interpretive form of the Run-Time Program.

Some of the uses of the LIST DIM command are:

- Generate a quick dump of variables during program debugging.
- Generate diagnostic information from end-user with application problems.

LIST DIM (cont.)

- Display current dimensions of variables which have been redimensioned using the MAT REDIM statement.
- Display current variable status, whether common or non-common variable is in use.

Examples:

```
:LIST DIM A$,S$(
:LIST DIM F$,L$
:LIST DIM * A,D

:0009 COM C$16
:0010 DIM A(10),B$(11)32
:0020 FOR I=1 TO 10
:    A(I)=I
:    B$(I)="ABC"
: NEXT I
:0030 C$="TEST LIST DIM  "
:RUN
:LIST DIM *
DIM A(10)
(1)         1
(2)         2
(3)         3
(4)         4
(5)         5
(6)         6
(7)         7
(8)         8
(9)         9
(10)        10
DIM B$(11)32
(1)         "ABC        "  HEX(4142 4320 2020 2020 2020 2020 2020 2020)
  STR(17)   "           "  HEX(2020 2020 2020 2020 2020 2020 2020 2020)
(2)         "ABC        "  HEX(4142 4320 2020 2020 2020 2020 2020 2020)
  STR(17)   "           "  HEX(2020 2020 2020 2020 2020 2020 2020 2020)
(3)         "ABC        "  HEX(4142 4320 2020 2020 2020 2020 2020 2020)
  STR(17)   "           "  HEX(2020 2020 2020 2020 2020 2020 2020 2020)
(4)         "ABC        "  HEX(4142 4320 2020 2020 2020 2020 2020 2020)
  STR(17)   "           "  HEX(2020 2020 2020 2020 2020 2020 2020 2020)
(5)         "ABC        "  HEX(4142 4320 2020 2020 2020 2020 2020 2020)
  STR(17)   "           "  HEX(2020 2020 2020 2020 2020 2020 2020 2020)
(6)         "ABC        "  HEX(4142 4320 2020 2020 2020 2020 2020 2020)
  STR(17)   "           "  HEX(2020 2020 2020 2020 2020 2020 2020 2020)
(7)         "ABC        "  HEX(4142 4320 2020 2020 2020 2020 2020 2020)
  STR(17)   "           "  HEX(2020 2020 2020 2020 2020 2020 2020 2020)
(8)         "ABC        "  HEX(4142 4320 2020 2020 2020 2020 2020 2020)
  STR(17)   "           "  HEX(2020 2020 2020 2020 2020 2020 2020 2020)
(9)         "ABC        "  HEX(4142 4320 2020 2020 2020 2020 2020 2020)
  STR(17)   "           "  HEX(2020 2020 2020 2020 2020 2020 2020 2020)
(10)        "ABC        "  HEX(4142 4320 2020 2020 2020 2020 2020 2020)
  STR(17)   "           "  HEX(2020 2020 2020 2020 2020 2020 2020 2020)
(11)        ALL(" ")       ALL(20)
COM C$16
        "TEST LIST DIM "   HEX(5445 5354 204C 4953 5420 4449 4D20 2020)
DIM I
        10
```

LIST DIM (cont.)

Compatibility Issues:

This statement is supported only with Release 2.0 or greater.

This statement is not valid in Wang 2200 Basic-2.

References:

COM
DIM STATIC
DIM PUBLIC
DIM RECURSIVE
LIST V
LIST STACK DIM Inspection and Modification of Program Logic - Section 6.3 of the Programmer's Guide
Inspection and Modification of Variables - Section 6.4 of the Programmer's Guide
Inspection and Modification of Environment - Section 6.6 of the Programmer's Guide

LIST DT

General Form:

```
LIST[title] [S] DT
            [F]
```

Where:

```
title = an optional descriptive title; must be a literal-string.

S     = specifies that a page break be performed.

F     = specifies that no page break be performed.
```

Discussion:

The LIST DT command lists various information about the current NPL environment. This information consists of:

- Device-addresses associated with current SELECT [R,G,D], ERROR, ROUND, P (pause), LINE, CI, INPUT, PLOT, TAPE, LOG entries in the Internal Device Table.

- Device-addresses associated with current SELECT PRINT, LIST, CO, and DISK entries in the Internal Device Table.

- File slot information for any cataloged files open for processing are displayed with the current disk address information for each file slot entry.

NOTE: **If file slots above #15 are defined (by SELECT DISK/FILE NUMBER), the highest file #slot is displayed even if no address is assigned. Refer to Section 7.2.3 of the Programmer's Guide for details on the Internal Device Table.**

- Device-addresses and their corresponding native operating system file or device defined in the Device Equivalence Table as established using $DEVICE statements. Refer to Section 7.2.2 of the Programmer's Guide for details on the Device Equivalence Table.

LIST DT (cont.)

- Program load sequence. This lists up to six NPL programs LOADed into memory since memory was last cleared. If more than six programs have been loaded, the program names displayed are those of the first program loaded plus the last five programs loaded.

  The $PROGRAM and "Program Load Sequence" information displayed in LIST DT applies to the current RUN module only, and does not show modules loaded using INCLUDE statements.

- If the program is halted and can be continued, a line is displayed showing the name of the Executing Module.

- The module name of the RUN module is displayed. On current releases, this is always blank.

- The module name of the LIST module is displayed. On current releases, this shows the current run module’s flag, in the same format as for INCLUDE modules.

- All modules (except the root module) which are currently loaded display a line in the format:

```
INCLUDE T/xxx, "FILENAME" TO "ModuleIdentifier"  :status
```

Here the T/xxx and FILENAME fields identify the device and filename used to initially load the module, and ModuleIdentifier is the internal module name.

The "status" field shows the current modification state of the module and is one of the following keywords:

LOAD	<-- module unchanged since loaded (using INCLUDE)
ERROR	<-- was loaded by INCLUDE but could not be resolved
MERGE	<-- module overlaid or otherwise modified since load
CLEAR	<-- module is clear of program lines
SAVE	<-- module unchanged since last saved (in full)

LIST DT (cont.)

The RUN indicator shows that the module is currently resolved (Public functions may be called either indirectly, or directly, if module is currently INCLUDEd).

The COM indicator shows that the module has defined common (COM) variables, and, so, is normally not deleted after a RUN, even when no longer referenced by INCLUDE statements.

- TRACE status. This shows the current trace status in effect.
- STEP status. This shows the current STEP status in effect including the STEP # range for the current LIST module if STEP is on. The name of the current LIST module in quotes (if not blank) precedes the STEP information.

Refer to LIST general parameters section for details on parameters common to all LIST statements.

LIST DT performs no operation on the non-interpretive form of the RunTime Program.

Examples:

```
:LIST DT
SELECT R, ERROR >60, ROUND, P, LINE 24, LISTLINE 55
SELECT CI /001, INPUT /001, PLOT /000, TAPE /000, LOG /000 OFF

SELECT PRINT /005(80), LIST /005(80), CO /005(80)

FILE ADDRESS FILE-NAME   START   CURRENT   END
       * NO OPEN FILES *

$DEVICE(/004) = "/dev/prn"
$DEVICE(/015) = "/dev/prn"
$DEVICE(/010) = "A:"
$DEVICE(/D11) = "platter1.bs2"
$DEVICE(/D12) = "platter2.bs2"
$DEVICE(/020) = "rivexp.bs2"

Program Load Sequence:START
Executing MODULE "rivfact"
RUN MODULE " "
LIST MODULE "rivfact"
INCLUDE T/020, "rivfact " TO "rivfact":MERGE :RUN
TRACE OFF
"rivfact"
STEP OFF
```

LIST DT (cont.)

Compatibility Issues:

The LIST DT statement is functionally the same in Wang 2200 Basic-2. However the command has been extended to include the NPL Device Equivalence Table, and in addition, the format of the output has been modified extensively.

LIST DT is supported on NPL Revisions 2.00 and greater.

Display of SELECT LOG device, SELECT LISTLINE value, TRACE status, and STEP status are new features implemented in Revision 3.0 of NPL.

Display of RUN MODULE, LIST MODULE and INCLUDEd modules are features implemented in Revision 4.0 of NPL.

References:

$DEVICE
SELECT
STEP
TRACE
Internal Device Table - Section 7.2.3 of the Programmer's Guide
Device Equivalence Table - Section 7.2.2 of the Programmer's Guide

LIST FIELD

General Form:

```
     LIST[title][S] FIELD[*] [([low-line][,[high-line]])]
             [F]
                             [name-1][,[name-2]]
```

Where:

title	= an optional descriptive title; must be a literal-string.
S	= specifies that a page break be performed.
F	= specifies that no page break be performed.
*	= list program statement as opposed to just line-number.
low-line	= lowest line-number for which to show references.
high-line	= highest line-number for which to show references.
name-1	= low range of field identifier to be displayed.
name-2	= high range of field identifier to be displayed.

Discussion:

The LIST FIELD command produces a listing of all FIELDs referenced by the program in the current LIST module, and on which program lines they are referenced.

The LIST function refers to program text in the current list module. This is set to the currently executing module whenever a program HALTs or continues, or when changed using the MODULE command, and can be referenced using LIST DT.

The default format for the LIST FIELD command lists line-numbers where the specified FIELD(s) appears. Specifying the "*" parameter causes the program statement which contains the specified FIELD(s) to be listed in addition to the line #.

LIST FIELD (cont.)

Refer to LIST general parameters for details on general parameters for all LIST statements.

The optional low/high range parameters are used to specify the range of lines accessed from which references are displayed. These operate as follows:

- If only low-line is specified, only that specific program line is accessed.
- If low-line, (comma) is specified, all program lines starting at low-line are accessed in ascending sequence.
- If ,(comma)high-line is specified, all program lines starting at the lowest ASCII sequence up to and including high-line are accessed.
- If low-line, high-line is specified, all program lines within the range of low-line to high-line inclusive are accessed.
- If no line-numbers are specified, the entire program in memory is accessed.

The optional name range parameters operate as follows:

- If only name-1 is specified, LIST FIELD output for only that specific field is generated.
- If name-1, (comma) is specified, LIST FIELD output for fields starting with name-1 in ascending ASCII sequence is generated.

```
:LIST FIELD (100,200)
:LIST FIELD(2000,) .Boats$,.Trucks$
:LIST FIELD * (,4000),.Hats
```

- If , (comma)name-2 is specified, LIST FIELD output for fields starting at the lowest ASCII sequence up to and including name-2 is generated.
- If name-1,name-2 is specified, LIST FIELD output for fields within the range of name-1 to name-2 inclusive is generated.
- If no range parameters are specified, LIST FIELD output is generated for all fields referenced by the program in the current LIST module.

LIST FIELD (cont.)

If exactly 1 type of field (numeric-scalar, alpha-scalar, numeric-array, alpha-array) is specified in name-1 and name-2, only fields of that type are listed. If different field types are specified, all field types are listed.

Field arrays are specified in a LIST FIELD statement using a special syntax. The array designator is specified, followed by an open parenthesis "(". For example, the arrays .Table$() and .Counters() would be specified by:

```
0010 LIST FIELD .Counters(,.Table$(
```

The primary difference between LIST FIELD and LIST STACK DIM is that LIST FIELD shows only fields referenced by the program in the current LIST module. LIST STACK DIM displays all fields in memory, in stack order, even if not referenced by the program in the current LIST module.

LIST FIELD performs no operation (NOP) on the non-interpretive form of the RunTime Program.

Examples:

```
:LIST FIELD (100,200)
:LIST FIELD(2000,) .Apples$,.Oranges
:LIST FIELD *(,4000),.Pages
:LIST FIELD
:LIST FIELD .Sticks,.Firewood$
:LIST FIELD * .Sticks,.Firewood$
:LIST FIELD .Units
:LIST FIELD *, .Firewood$

                    SAMPLE PROGRAM
0010 RECORD /PUBLIC Area
   :   FIELD LeftUpperQuad=HEX(5202)
   :   FIELD LeftLowerQuad=HEX(5202)
   :   FIELD RightUpperQuad=HEX(5202)
   :   FIELD RightLowerQuad=HEX(5202)
   : END RECORD
0020 DIM Buffer$#RECORDLENGTH(Area)
0030 Buffer$.LeftUpperQuad=2.6
0040 Buffer$.LeftLowerQuad=3.8
:

:
:LIST FIELD
.LeftLowerQuad
    - 0010 0040
.LeftUpperQuad
    - 0010 0030
.RightLowerQuad
    - 0010
.RightUpperQuad
    - 0010
```

LIST FIELD (cont.)

```
:LIST FIELD *

.LeftLowerQuad--------------------------------------------------------
-
0010 :: FIELD LeftLowerQuad=HEX(5202)
0040 Buffer$.LeftLowerQuad=3.8

.LeftUpperQuad--------------------------------------------------------
0010 : FIELD LeftUpperQuad=HEX(5202)
0030 Buffer$.LeftUpperQuad=2.6

.RightLowerQuad-------------------------------------------------------
0010 :::: FIELD RightLowerQuad=HEX(5202)

.RightUpperQuad-------------------------------------------------------
0010 ::: FIELD RightUpperQuad=HEX(5202)
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

FIELD
RECORD
Inspection and Modification of Program Text - Section 6.5 of the Programmer's Guide

LIST FUNCTION

General Form:

```
LIST[title][S] FUNCTION[*] [([low-line][,[high-line]])]
           [F]
           [function-identifier1][,[function-identifier2]]
```

Where:

```
title                 = an optional descriptive title; must be a
                        literal-string.

S                     = specifies that a page break be performed.

F                     = specifies that no page break be performed.

*                     = list program statement as opposed to just
                        line-number.

low-line              = lowest line-number for which to show refer-
                        ences.

high-line             = highest line-number for which to show ref-
                        erences.

function-identifier1 = low range of function to be displayed.

function-identifier2 = high range of function to be displayed.
```

Discussion:

The LIST FUNCTION produces a listing of all functions referenced by the program in the current LIST module, and on which program lines they are referenced.

The LIST function refers to program text in the current list module. This is set to the currently executing module whenever a program HALTs or continues, or when changed using the MODULE command, and can be referenced using LIST DT.

LIST FUNCTION (cont.)

The default format for the LIST FUNCTION command lists line-numbers where the specified function references appear. Specifying the "*" parameter causes the program statement containing the specified FUNCTION(s) to be listed in addition to the line number.

Refer to LIST general parameters for details on general parameters for all LIST statements.

The optional name range parameters operate as follows:

- If only function-identifier1 is specified, LIST FUNCTION output for only that specific function is generated.

- If function-identifier1, (comma) is specified, LIST FUNCTION output for functions starting with function-identifier2 in ascending ASCII sequence is generated.

```
:LIST FUNCTION (100,200)
:LIST FUNCTION (2000,) 'Get_Status$,.'Current_Status$
:LIST FUNCTION *(,4000),.'Count_Hats
```

- If , (comma)function_identifier2 is specified, LIST FUNCTION output for functions starting at the lowest ASCII sequence up to and including function-identifier2 is generated.

- If function-identifier1,function-identifier2 is specified, LIST FUNCTION output for functions within the range of function-identifier1 to function-identifier2 inclusive is generated.

- If no range parameters are specified, LIST FUNCTION output is generated for all functions referenced by the program in the current LIST module.

If exactly 1 type of function (numeric-scalar, alpha-scalar) is specified in function-identifier1 and function-identifier2, only functions of that type are listed. If different function types are specified, all function types are listed.

LIST FUNCTION (cont.)

The optional low/high range parameters are used to specify the range of lines accessed from which references are displayed. These operate as follows:

- If only low-line is specified, only that specific program line is accessed.
- If low-line, (comma) is specified, all program lines starting at low-line are accessed in ascending sequence.
- If ,(comma)high-line is specified, all program lines starting at the lowest ASCII sequence up to and including high-line are accessed.
- If low-line, high-line is specified, all program lines within the range of low-line to high-line inclusive are accessed.
- If no line-numbers are specified, the entire program in the current LIST module is accessed.

Examples:

```
:LIST FUNCTION
:LIST FUNCTION 'Do_it,'Did_it
:LIST FUNCTION * 'Do_it,'Did_it$
:LIST FUNCTION 'Get_Activity,
:LIST FUNCTION *,'Rain_Event

========================================R     0010 ;SAMPLE ROUTINES
0020 ;MAINLINE
0030 INCLUDE T "FUNCTS"
0040 Y$='Get_Position$(Z$)
========================================

========================================
0010 ;FUNCTS
0100 FUNCTION 'Get_Position$(A$16) /PUBLIC
0110 RETURN (A$)
0120 END FUNCTION
0200 FUNCTION 'Do_It(V)
0210 RETURN (V)
0220 END FUNCTION
========================================

:RUN
:
:LIST FUNCTION

 'Get_Position$
 - 0040
```

LIST FUNCTION (cont.)

```
:LIST FUNCTION *

’Get_Position$--------------------------------------------------------
0040 Y$=’Get_Position$(Z$)
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

LIST
LIST DT
MODULE
Inspection and Modification of Program Text - Section 6.5 of the Programmer’s Guide

LIST PROCEDURE

General Form:

```
     LIST[title][S] PROCEDURE[*] [([low-line][,[high-line]])]
               [F]
              [procedure-identifier1][,[procedure-identifier2]]
```

Where:

```
title                 = an optional descriptive title; must be a
                        literal-string.

S                     = specifies that a page break be performed.

F                     = specifies that no page break be performed.

*                     = list program statement as opposed to just
                        line-number.

low-line              = lowest line-number for which to show refer-
                        ences.

high-line             = highest line-number for which to show ref-
                        erences.

procedure-identifier1 = low range of procedure to be displayed.

procedure-identifier2 = high range of procedure to be displayed.
```

Discussion:

The LIST PROCEDURE command produces a listing of all PROCEDUREs referenced by the program in the current LIST module, and on which program lines they are referenced.

LIST PROCEDURE (cont.)

The LIST procedure refers to program text in the current list module. This is set to the currently executing module whenever a program HALTs or continues, or when changed using the MODULE command, and can be referenced using LIST DT.

The default format for the LIST PROCEDURE command lists line-numbers where the specified PROCEDURE(s) appears. Specifying the "*" parameter causes the program statement containing the specified PROCEDURE(s) to be listed in addition to the line number..

Refer to LIST general parameters for details on general parameters for all LIST statements.

The optional name range parameters operate as follows:

- If only procedure-identifier1 is specified, LIST PROCEDURE output for only that specific procedure is generated.
- If procedure-identifier1, (comma) is specified, LIST PROCEDURE output for procedures starting with procedure-identifier2 in ascending ASCII sequence is generated.

```
:LIST PROCEDURE (100,200)
:LIST PROCEDURE(2000,) 'Get_Status$,.'Current_Status$
:LIST PROCEDURE *(,4000),.'Count_Hats
```

- If , (comma)procedure-identifier2 is specified, LIST PROCEDURE output for procedures starting at the lowest ASCII sequence up to and including procedure-identifier2 is generated.
- If procedure-identifier1,procedure-identifier2 is specified, LIST PROCEDURE output for procedures within the range of procedure-identifier1 to procedure-identifier2 inclusive is generated.
- If no range parameters are specified, LIST PROCEDURE output is generated for all procedures referenced by the program in the current LIST module.

LIST PROCEDURE (cont.)

The optional low/high range parameters are used to specify the range of lines accessed from which references are displayed. These operate as follows:

- If only low-line is specified, only that specific program line is accessed.
- If low-line, (comma) is specified, all program lines starting at low-line are accessed in ascending sequence.
- If ,(comma)high-line is specified, all program lines starting at the lowest ASCII sequence up to and including high-line are accessed.
- If low-line, high-line is specified, all program lines within the range of low-line to high-line inclusive are accessed.
- If no line-numbers are specified, the entire program in the current list module is accessed.

Examples:

```
    :LIST PROCEDURE
    :LIST PROCEDURE 'Did_it,'Do_it
    :LIST PROCEDURE * 'Did_it,'Do_it$
    :LIST PROCEDURE 'Update_Activity,
    :LIST PROCEDURE *,'Rain_Event
0010 ; Mainline
    :PROCEDURE 'Set_Position(A$16) /PUBLIC /FORWARD
    :PROCEDURE 'Do_It( n ) /FORWARD
    :DIM n, Z$16
    :'Do_it( n )
    :'Set_Position(Z$)
0020 END
0030 PROCEDURE 'Set_Position( A$16 ) /BEGINS
    :RETURN
    :END PROCEDURE
    :PROCEDURE 'DoIt( n ) /BEGINS
    :RETURN
    :END PROCEDURE

:list procedure
'Do_It- 0010 0010 0030
'Set_Position
      - 0010 0010 0030

:list procedure *

'Do_It-----------------------------------------------
00100 :; PROCEDURE 'Do_It(N) /FORWARD
0010 ::::: 'Do_It(N)
0030 ::: PROCEDURE 'Do_It(N) /BEGINS
```

LIST PROCEDURE (cont.)

```
’Set_Position-----------------------------------------
0010 : PROCEDURE ’Set_Position(A$) /PUBLIC /FORWARD
0010 :::: ’Set_Position(Z$)
0030 PROCEDURE ’Set_Position(A$16) /BEGINS
 LIST PROCEDURE (cont.)
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

Inspection and Modification of Program Text - Section 6.5 of the Programmer’s Guide

LIST PUBLIC DEFFN

General Form:

```
LIST PUBLIC[title][S] DEFFN[*] [([low-line][,[high-line]])]
                   [F]
                   [DEFFN-identifier1][,[DEFFN-identifier2]]
```

Where:

```
title               = an optional descriptive title; must be a
                      literal-string.

S                   = specifies that a page break be performed.

F                   = specifies that no page break be performed.

*                   = list program statement as opposed to just
                      line-number.

low-line            = lowest line-number for which to show refer-
                      ences.

high-line           = highest line-number for which to show ref-
                      erences.

DEFFN-identifier1 = low range of DEFFN to be displayed.

DEFFN-identifier2 = high range of DEFFN to be displayed.
```

Discussion:

The LIST PUBLIC DEFFN command produces a listing of all DEFFNs declared by all currently loaded and resolved PUBLIC sections in the workspace. This allows selection of new public names which are not in conflict with currently loaded declarations. It also allows a review of the declared parameter names and types as a reminder when programming.

LIST PUBLIC DEFFN (cont.)

NOTE: The module in which the public section is declared must already be INCLUDEd, either by a previous program RUN or by an immediate mode "INCLUDE" statement.

A line-number range, if specified, is ignored for all LIST PUBLIC statements.

Refer to LIST general parameters for details on general parameters for all LIST statements.

The optional variable range parameters operate as follows:

- If only DEFFN-identifier1 is specified, LIST PUBLIC DEFFN output for only that specific DEFFN is generated.
- If DEFFN-identifier1, (comma) is specified, LIST PUBLIC DEFFN output for DEFFNs starting starting with DEFFN-identifier1 in ascending ASCII sequence is generated.

```
:LIST PUBLIC DEFFN 'Set_Length, 'Set_Width
:LIST PUBLIC DEFFN *,'Set_Count
```

- If , (comma)DEFFN-identifier2 is specified, LIST PUBLIC DEFFN output for DEFFNs starting at the lowest ASCII sequence up to and including DEFFN-identifier2 is generated.
- If DEFFN-identifier1,DEFFN-identifier2 are specified, LIST PUBLIC DEFFN output for variables within the range of DEFFN-identifier1 to DEFFN-identifier2 inclusive is generated.
- If no range parameters are specified, LIST PUBLIC DEFFN output is generated for all DEFFNs currently loaded and resolved PUBLIC sections in memory.

LIST PUBLIC DEFFN (cont.)

The F, S and * options are all permitted on this LIST statements.

The output of the statement shows the name of the indicated DEFFN, preceded by the module name in which the DEFFN is declared. If a "*" option is used, additional information about the DEFFN also appears in the listing. The extra information printed when the "*" option is used consists of the statement in which the declaration occurs.

NOTE: This public declaration statement is displayed even if the module in which it appears is scramble-protected.

No LIST PUBLIC examples should use a line number range.

Examples:

```
:LIST PUBLIC DEFFN
:LIST PUBLIC DEFFN 'Did_it,'Do_it
:LIST PUBLIC DEFFN * 'Did_it,'Do_it
:LIST PUBLIC DEFFN 'Set_activity,
:LIST PUBLIC DEFFN *,'Rain_Event


========================================
0010 ;SAMPLE ROUTINES
0020 ;MAINLINE
0025 DIM /PUBLIC A,B
0030 INCLUDE T "DEFFNS"
0040 GOSUB 'Calc_It(X,Y,Z)
========================================
0010 ;DEFFNS
0020 DEFFN  'Calc_It(A,B,C) /PUBLIC
0030 B=2: C=3: A=B*C+5
0035 PRINT A,B,C;  " Values in the DEFFN'"
0040 RETURN
0100 DEFFN 'Nothing(G)
0110 RETURN
========================================

:RUN
:
:LIST PUBLIC DEFFN

"DEFFNS"  DEFFN 'Calc_It

:LIST PUBLIC DEFFN *

"DEFFN" DEFFN 'Calc_It------------------------------------------------
0020 DEFFN 'Calc_It(A,B,C)/PUBLIC
```

LIST PUBLIC DEFFN (cont.)

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

Inspection and Modification of Program Text - Section 6.5 of the Programmer's Guide
LIST '

LIST PUBLIC FIELD

General Form:

```
LIST PUBLIC[title][S] FIELD[*] [([low-line][,[high-line]])]
               [F]
                                [name-1][,[name-2]]
```

Where:

```
title      = an optional descriptive title; must be a literal-
             string.

S          = specifies that a page break be performed.

F          = specifies that no page break be performed.

*          = list program statement as opposed to just line-num-
             ber.

name-1     = low range of field to be displayed.

name-2     = high range of field to be displayed.
```

Discussion:

The LIST PUBLIC FIELD command produces a listing of all FIELDs declared by all currently loaded and resolved PUBLIC sections in the workspace. This allows selection of new public names which are not in conflict with currently loaded declarations.

NOTE: The module in which the public section is declared must already be INCLUDEd, either by a previous program RUN or by an immediate mode "INCLUDE" statement.

A line-number range, if specified, is ignored for all LIST PUBLIC statements.

LIST PUBLIC FIELD (cont.)

Refer to LIST general parameters for details on general parameters for all LIST statements.

The optional name range parameters operate as follows:

- If only name-1 is specified, LIST PUBLIC FIELD output for only that specific field is generated.
- If name-1, (comma) is specified, LIST PUBLIC FIELD output for fields starting with name-1 in ascending ASCII sequence is generated.

  ```
  :LIST PUBLIC FIELD
  :LIST PUBLIC FIELD .Boats$,.Trucks$
  :LIST PUBLIC FIELD *,.Hats
  ```
- If , (comma)name-2 is specified, LIST PUBLIC FIELD output for fields starting at the lowest ASCII sequence up to and including name-2 is generated.
- If name-1,name-2 is specified, LIST PUBLIC FIELD output for fields within the range of name-1 to name-2 inclusive is generated.
- If no range parameters are specified, LIST PUBLIC FIELD output is generated for all fields declared by all currently loaded and resolved public sections in the workspace.

If exactly 1 type of field (numeric-scalar, alpha-scalar, numeric-array, alpha-array) is specified in name-1 and name-2, only fields of that type are listed. If different field types are specified, all field types are listed.

Field arrays are specified in a LIST PUBLIC FIELD statement using a special syntax. The array designator is specified, followed by an open parenthesis "(". For example, the arrays .Table$() and .Counters() would be specified by:

```
0010 LIST PUBLIC FIELD .Table$(, .Counters(
```

LIST PUBLIC FIELD performs no operation on the non-interpretive form of the Run-Time Program.

LIST PUBLIC FIELD (cont.)

The F, S and * options are all permitted on this LIST statements. However, specifying a restricted line range has no effect on the output.

The output of the statement shows the name of the field, preceded by the module name in which the variable is declared. If a "*" option is used, additional information about the field also appears in the listing. The extra information field consists of the #FIELD-START, #FIELDLENGTH and $FIELDFORMAT function values.

NOTE: This public declaration statement is displayed even if the module in which it appears is scramble-protected.

Examples:

```
:LIST PUBLIC FIELD .Apples$,.Oranges
:LIST PUBLIC FIELD *,.Pages
:LIST PUBLIC FIELD
:LIST PUBLIC FIELD .Firewood$,.Sticks
:LIST PUBLIC FIELD * .Firewood$,.Sticks,
:LIST PUBLIC FIELD .Units
: LIST PUBLIC FIELD *, .Firewood$

                    SAMPLE PROGRAM
0005 INCLUDE T "RECORD"
0010 RECORD /PUBLIC Area
   :   FIELD LeftUpperQuad=HEX(5202)
   :   FIELD LeftLowerQuad=HEX(5202)
   :   FIELD RightUpperQuad=HEX(5202)
   :   FIELD RightLowerQuad=HEX(5202)
   : END RECORD

:
: RUN
:
:LIST PUBLIC FIELD
"RECORD" DIM /PUBLIC FIELD .LeftLowerQuad
"RECORD" DIM /PUBLIC FIELD .LeftUpperQuad
"RECORD" DIM /PUBLIC FIELD .RightLowerQuad
"RECORD" DIM /PUBLIC FIELD .RightUpperQuad
```

LIST PUBLIC FIELD (cont.)

```
:LIST PUBLIC FIELD *

"RECORD" DIM /PUBLIC FIELD .LeftLowerQuad------------------------------
#FIELDSTART()=3 #FIELDLENGTH()=2
$FIELDFORMAT()=HEX(5202)

"RECORD" DIM /PUBLIC FIELD .LeftUpperQuad------------------------------
#FIELDSTART()=1 #FIELDLENGTH()=2
$FIELDFORMAT()=HEX(5202)

"RECORD" DIM /PUBLIC FIELD .RightLowerQuad-----------------------------
-----------
#FIELDSTART()=7 #FIELDLENGTH()=2
$FIELDFORMAT()=HEX(5202)


"RECORD" DIM /PUBLIC FIELD .RightLowerQuad-----------------------------
-----------
#FIELDSTART()=5 #FIELDLENGTH()=2
$FIELDFORMAT()=HEX(5202)
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

LIST RECORD
LIST FIELD
Inspection and Modification of Program Text - Section 6.5 of the Programmer's Guide

LIST PUBLIC FUNCTION

General Form:

```
LIST PUBLIC[title][S] FUNCTION[*] [([low-line][,[high-line]])]
           [F]
           [function-identifier1][,[function-identifier2]]
```

Where:

```
title                 = an optional descriptive title; must be a
                        literal-string.

S                     = specifies that a page break be performed.

F                     = specifies that no page break be performed.

*                     = list program statement as opposed to just
                        line-number.

function-identifier1 = low range of function to be displayed.

function-identifier2 = high range of function to be displayed.
```

Discussion:

The LIST PUBLIC FUNCTION command produces a listing of all FUNCTIONs declared by all currently loaded and resolved PUBLIC sections in the workspace. This allows selection of new public names which are not in conflict with currently loaded declarations. It also allows a review of the declared parameter names and types as a reminder when programming.

NOTE: The module in which the public section is declared must already be INCLUDEd, either by a previous program RUN or by an immediate mode "INCLUDE" statement.

LIST PUBLIC FUNCTION (cont.)

A line-number range, if specified, is ignored for all LIST PUBLIC statements.

Refer to LIST general parameters for details on general parameters for all LIST statements.

The optional function name range parameters operate as follows:

- If only function-identifier1 is specified, LIST PUBLIC FUNCTION output for only that specific function is generated.
- If function-identifier1, (comma) is specified, LIST PUBLIC FUNCTION output for functions starting starting with function-identifier1 in ascending ASCII sequence is generated.

```
:LIST PUBLIC FUNCTION 'Get_Length$, 'Get_Width$
:LIST PUBLIC FUNCTION *,'Get_Box_Count
```

- If , (comma)function-identifier2 is specified, LIST PUBLIC FUNCTION output for functions starting at the lowest ASCII sequence up to and including function-identifier2 is generated.
- If function-identifier1,function-identifier2 are specified, LIST PUBLIC FUNCTION output for variables within the range of function-identifier1 to function-identifier2 inclusive is generated.
- If no range parameters are specified, LIST PUBLIC FUNCTION output is generated for all variables declared by the program currently in memory.

If exactly 1 type of function (numeric-scalar, alpha-scalar) is specified in function-identifier1 and function-identifier2, only functions of that type are listed. If different function types are specified, all function types are listed.

The F, S and * options are all permitted on this LIST statements.

LIST PUBLIC FUNCTION (cont.)

The output of the statement shows the name of the indicated variable type, preceded by the module name in which the function is declared. If a "*" option is used, additional information about the function also appears in the listing. The extra information consists of the statement in which the declaration occurs.

NOTE: This public declaration statement is displayed even if the module in which it appears is scramble-protected.

When a function has both a /FORWARD and a subsequent declaration, only one is displayed.

The first declaration (usually /FORWARD) is displayed (including any embedded inline comments).

Examples:

```
:LIST PUBLIC FUNCTION
:LIST PUBLIC FUNCTION ,'Did_it,'Do_it
:LIST PUBLIC FUNCTION *'Did_it$, 'Do_it
:LIST PUBLIC FUNCTION 'Get_activity,
:LIST PUBLIC FUNCTION *,'Weather_event


=======================================
0010 ;SAMPLE ROUTINES
0020 ;MAINLINE
0030 INCLUDE T "FUNCTS"
0040 Y$='Get_Position$(Z$)
========================================


========================================
0010 ;FUNCTS
0100 FUNCTION 'Get_Position$
0110 RETURN (A$)
0120 END FUNCTION
0200 FUNCTION 'Do_It(V)
0210 RETURN (V)
0220 END FUNCTION
========================================

:RUN
:
:LIST PUBLIC FUNCTION

"FUNCTS" 'Get_Position$

:LIST PUBLIC FUNCTION *

"FUNCTS" 'Get_Position$-------------------------------------------------
0100 FUNCTION 'Get_Position$(A$16)/PUBLIC
```

LIST PUBLIC FUNCTION (cont.)

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

Inspection and Modification of Program Text - Section 6.5 of the Programmer's Guide

LIST PUBLIC PROCEDURE

General Form:

```
LIST PROCEDURE[title][S] PROCEDURE[*] [([low-line][,[high-line]])]
                        [F]
             [procedure-identifier1][,[procedure-identifier2]]
Where:

title        = an optional descriptive title; must be a literal-
               string.

S            = specifies that a page break be performed.

F            = specifies that no page break be performed.

*            = list program statement as opposed to just line-num-
               ber.

identifier1 = low range of procedure to be displayed.

identifier2 = high range of procedure to be displayed.
```

Discussion:

The LIST PUBLIC PROCEDURE command produces a listing of all procedures declared by all currently loaded and resolved PUBLIC sections in the workspace. This allows selection of new public names which are not in conflict with currently loaded declarations. It also allows a review of the declared parameter names and types as a reminder when programming.

NOTE: The module in which the public section is declared must already be INCLUDEd, either by a previous program RUN or by an immediate mode "INCLUDE" statement.

LIST PUBLIC PROCEDURE (cont.)

A line-number range, if specified, is ignored for all LIST PUBLIC statements.

Refer to LIST general parameters for details on general parameters for all LIST statements.

The optional variable range parameters operate as follows:

- If only procedure-identifier1 is specified, LIST PUBLIC PROCEDURE output for only that specific procedure is generated.

- If procedure-identifier1, (comma) is specified, LIST PUBLIC PROCEDURE output for procedures starting starting with procedure-identifier1 in ascending ASCII sequence is generated.

```
:LIST PUBLIC PROCEDURE 'Set_Length,'Set_Width
:LIST PUBLIC PROCEDURE *,'Set_Box_Count
```

- If , (comma)procedure-identifier2 is specified, LIST PUBLIC PROCEDURE output for procedures starting at the lowest ASCII sequence up to and including procedure-identifier2 is generated.

- If procedure-identifier1,procedure-identifier2 are specified, LIST PUBLIC PROCEDURE output for variables within the range of procedure-identifier1 to procedure-identifier2 inclusive is generated.

- If no range parameters are specified, LIST PUBLIC PROCEDURE output is generated for all variables declared by the program currently in memory.

The F, S and * options are all permitted on this LIST statement.

The output of the statement shows the name of the indicated procedure type, preceded by the module name in which the procedure is declared. If a "*" option is used, additional information about the procedure appears in the listing. The extra information consists of the statement in which the declaration occurs.

LIST PUBLIC PROCEDURE (cont.)

NOTE: This public declaration statement is displayed even if the module in which it appears is scramble-protected.

When a procedure has both a /FORWARD and a subsequent declaration, only one is displayed.

The first declaration (usually /FORWARD) is displayed (including any embedded inline comments).

Examples:

```
:LIST PUBLIC PROCEDURE
:LIST PUBLIC PROCEDURE 'Did_it,'Do_it
:LIST PUBLIC PROCEDURE * 'Did_it'Do_it
:LIST PUBLIC PROCEDURE 'Set_Activity,
:LIST PUBLIC PROCEDURE *,'Rain_Event

=======================================
0010 ;SAMPLE ROUTINES
0020 ;MAINLINE
0030 INCLUDE T "PROCS"
0040 'Set_Position(Z$)
========================================

========================================
0010 ;PROCS
0100 PROCEDURE 'Set_Position(A$16) /PUBLIC
0110 RETURN
0120 END PROCEDURE
0200 procedure 'Do_It(V)
0210 RETURN (V)
0220 END PROCEDURE
========================================

:RUN
:
:LIST PUBLIC PROCEDURE

"PROCS" 'Set_Position

:LIST PUBLIC PROCEDURE *

"PROCS" 'Set_Position-------------------------------------------------
0100 PROCEDURE 'Set_Position(A$16)/PUBLIC
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

Inspection and Modification of Program Text - Section 6.5 of the Programmer's Guide
LIST PROCEDURE

LIST PUBLIC RECORD

General Form:

```
LIST PUBLIC[title][S] RECORD[*] [([low-line][,[high-line]])]
           [F]
           [name-1][,name-2]
```

Where:

```
title     = an optional descriptive title; must be a literal-
            string.

S         = specifies that a page break be performed.

F         = specifies that no page break be performed.

*         = list program statement as opposed to just line-num-
            ber.

name-1    = low range of records to be displayed.

name-2    = high range of records to be displayed.

low-line  = lowest line-number for which to show references.

high-line = highest line-number for which to show references.
```

Discussion:

The LIST PUBLIC RECORD command produces a listing of all RECORD(s) declared by all currently loaded and resolved PUBLIC sections in the workspace. This allows selection of new public names which are not in conflict with currently loaded declarations. It also allows a review of the declared names as a reminder when programming.

NOTE: The module in which the public section is declared must already be INCLUDEd, either by a previous program RUN or by an immediate mode "INCLUDE" statement.

LIST PUBLIC RECORD (cont.)

A line-number range, if specified, is ignored for all LIST PUBLIC statements.

Refer to LIST general parameters for details on general parameters for all LIST statements.

The optional name range parameters operate as follows:

- If only name-1 is specified, LIST PUBLIC RECORD output for only that specific record is generated.
- If name-1, (comma) is specified, LIST PUBLIC RECORD output for records starting with name-1 in ascending ASCII sequence is generated.

```
:LIST PUBLIC RECORD
:LIST PUBLIC RECORD Employee,Payroll
:LIST PUBLIC RECORD *,.Hats
```

- If , (comma)name-2 is specified, LIST PUBLIC RECORD output for records starting at the lowest ASCII sequence up to and including name-2 is generated.
- If name-1,name-2 is specified, LIST PUBLIC RECORD output for records within the range of name-1 to name-2 inclusive is generated.
- If no range parameters are specified, LIST PUBLIC RECORD output is generated for all records declared by all currently loaded and resolved public sections in the workspace.

LIST PUBLIC RECORD performs no operation (NOP) on the non-interpretive form of the RunTime Program.

The F, S and * options are all permitted on this LIST statements.

The output of the statement shows the name of the indicated record, preceded by the module name in which the variable is declared. If a "*" option is used, additional information about the record appears in the listing. The extra information consists of the #RECORDLENGTH function value.

LIST PUBLIC RECORD (cont.)

NOTE: This public declaration statement is displayed even if the module in which it appears is scramble-protected.

Examples:

```
:LIST PUBLIC RECORD Apples,Oranges
:LIST PUBLIC RECORD *,Birds
:LIST PUBLIC RECORD
:LIST PUBLIC RECORD Firewood,Sticks
LIST PUBLIC RECORD * Firewood,Sticks
LIST PUBLIC RECORD Units
 LIST PUBLIC RECORD *, Firewood

                   MODULE 1
0010 ;MYMAIN
0020 INCLUDE T "RECORD"
0030 DIM Buffer$#RECORDLENGTH(Area)
0040 Buffer$.LeftUpperQuad=3.5

                    MODULE 2
0010 RECORD /PUBLIC Area
   :   FIELD LeftUpperQuad=HEX(5202)
   :    FIELD LeftLowerQuad=HEX(5202)
   :   FIELD RightUpperQuad=HEX(5202)
   :   FIELD RightLowerQuad=HEX(5202)
   : END RECORD
:
: RUN
:
:LIST PUBLIC RECORD
:"RECORD" DIM /PUBLIC RECORD Area

:LIST PUBLIC RECORD *

"RECORD" DIM /PUBLIC RECORD Area------------------------------------
#RECORDLENGTH()=8
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

LIST RECORD
LIST FIELD
Inspection and Modification of Program Text - Section 6.5 of the Programmer's Guide

LIST PUBLIC V

General Form:

```
     LIST PUBLIC[title][S] V[*] [([low-line][,[high-line]])]
                [F]
                                [variable-1][,[variable-2]]
```

Where:

```
     title      = an optional descriptive title; must be a literal-
                  string.

     S          = specifies that a page break be performed.

     F          = specifies that no page break be performed.

     *          = list program statement as opposed to just line-num-
                  ber.

     variable-1 = low range of variable to be displayed.

     variable-2 = high range of variable to be displayed.

     low-line   = lowest line-number for which to show references.

     high-line  = highest line-number for which to show references.
```

Discussion:

The LIST PUBLIC V command produces a listing of all public variables declared by all currently loaded and resolved PUBLIC sections in the workspace. This allows selection of new public names which are not in conflict with currently loaded declarations. It also allows a review of the declared variables and types as a reminder when programming.

LIST PUBLIC V (cont.)

NOTE: The module in which the public section is declared must already be INCLUDEd, either by a previous program RUN or by an immediate mode "INCLUDE" statement.

A line-number range, if specified, is ignored for all LIST PUBLIC statement.

Refer to LIST general parameters for details on general parameters for all LIST statements.

The optional name range parameters operate as follows:

- If only variable-1 is specified, LIST PUBLIC V output for only that specific variable is generated.
- If variable-1, (comma) is specified, LIST PUBLIC V output for variables starting with variable-1 in ascending ASCII sequence is generated.

```
:LIST PUBLIC V
:LIST PUBLIC V Boats$,Trucks$
:LIST PUBLIC V *,Hats
```

- If , (comma)variable-2 is specified, LIST PUBLIC V output for variables starting at the lowest ASCII sequence up to and including variable-2 is generated.
- If variable-1,variable-2 is specified, LIST PUBLIC V output for variables within the range of variable-1 to variable-2 inclusive is generated.
- If no range parameters are specified, LIST PUBLIC V output is generated for all variables declared by all currently loaded and resolved public sections in the workspace.

If exactly 1 type of variable (numeric-scalar, alpha-scalar, numeric-array, alpha-array) is specified in variable-1 and variable-2, only variables of that type are listed. If different variable types are specified, all variables types are listed.

Arrays are specified in a LIST PUBLIC V statement using a special syntax. The array designator is specified, followed by an open parenthesis "(". For example, the arrays Table$() and Counters() would be specified by:

```
0010 LIST PUBLIC V Table$(, Counters(
```

LIST PUBLIC V (cont.)

LIST PUBLIC V performs no operation (NOP) on the non-interpretive form of the Run-Time Program.

The F, S and * options are all permitted on this LIST statements.

The output of the statement shows the name of the indicated variable, preceded by the module name in which the variable is declared. If a "*" option is used, additional information about the variable also appears in the listing. The extra information consists of the current value of the variable.

Examples:

```
0020 PUBLIC
0030 X,Y,Z$10
0040 END PUBLIC

:LIST PUBLIC V
" " DIM /PUBLIC X
" " DIM /PUBLIC Y
" " DIM /PUBLIC Z$10

:LIST PUBLIC V *
" " DIM /PUBLIC X-------------------------------------------------
              0
" " DIM /PUBLIC Y-------------------------------------------------
               0
" " DIM /PUBLIC Z$------------------------------------------------
                "                " HEX(2020 2020 2020 2020 2020)
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

Inspection and Modification of Program Text - Section 6.5 of the Programmer's Guide

LIST RECORD

General Form:

```
     LIST[title][S] RECORD[*] [([low-line][,[high-line]])]
                [F]
                               [name-1][,[name-2]]
```

Where:

title	= an optional descriptive title; must be a literal-string.
S	= specifies that a page break be performed.
F	= specifies that no page break be performed.
*	= list program statement as opposed to just line-number.
low-line	= lowest line-number for which to show references.
high-line	= highest line-number for which to show references.
name-1	= low range of record identifier to be displayed.
name-2	= high range of record identifier to be displayed.

Discussion:

The LIST RECORD command produces a listing of all RECORD(s) declared by the program in the current LIST module, and what program lines they are declared on.

LIST RECORD (cont.)

The LIST function refers to program text in the current list module. This is set to the currently executing module whenever a program HALTs or continues, or when changed using the MODULE command, and can be declared using LIST DT.

The default format for the LIST RECORD command lists line-numbers where the specified RECORD(s) appears. Specifying the "*" parameter causes the program statement which contains the specified RECORD(s) to be listed in addition to the line #.

Refer to LIST general parameters for details on general parameters for all LIST statements.

The optional low/high range parameters are used to specify the range of lines accessed from which references are displayed. These operate as follows:

- If only low-line is specified, only that specific program line is accessed.
- If low-line, (comma) is specified, all program lines starting at low-line are accessed in ascending sequence.
- If ,(comma)high-line is specified, all program lines starting at the lowest ASCII sequence up to and including high-line are accessed.
- If low-line, high-line is specified, all program lines within the range of low-line to high-line inclusive are accessed.
- If no line-numbers are specified, the entire program in the current list module is accessed.

LIST RECORD performs no operation (NOP) on the non-interpretive form of the Run-Time Program.

The optional name range parameters operate as follows:

- If only name-1 is specified, LIST RECORD output for only that specific RECORD is generated.
- If name-1, (comma) is specified, LIST RECORD output for RECORDS starting with name-1 in ascending ASCII sequence is generated.

LIST RECORD (cont.)

```
:LIST RECORD (100,200)
:LIST RECORD (2000,) Boats,Trucks
:LIST RECORD *(,4000),Hats
```

- If , (comma)name-2 is specified, LIST RECORD output for records starting at the lowest ASCII sequence up to and including name-2 is generated.
- If name-1,name-2 is specified, LIST RECORD output for records within the range of name-1 to name-2 inclusive is generated.
- If no range parameters are specified, LIST RECORD output is generated for all records declared by the program in the current LIST module.

Examples:

```
:LIST RECORD Apples,Oranges
:LIST RECORD *,Birds
:LIST RECORD
:LIST RECORD Firewood,Sticks
LIST RECORD * Firewood,Sticks
LIST RECORD Units
: LIST RECORD *, Firewood

                   MODULE 1
0010 ;MYMAIN
0020 INCLUDE T "RECORD"
0030 DIM Buffer$#RECORDLENGTH(Area)
0040 Buffer$.LeftUpperQuad=3.5

                    MODULE 2
0010 RECORD /PUBLIC Area
   :   FIELD LeftUpperQuad=HEX(5202)
   :   FIELD LeftLowerQuad=HEX(5202)
   :   FIELD RightUpperQuad=HEX(5202)
   :   FIELD RightLowerQuad=HEX(5202)
   : END RECORD

:RUN
:
:LIST RECORD
:Area - 0030

:LIST RECORD *

"Area--------------------------------------------------------------
0030 DIM Buffer$.#RECORDLENGTH(Area)
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

LIST RECORD (cont.)

References:

LIST PUBLIC RECORD
LIST FIELD
Inspection and Modification of Program Text - Section 6.5 of the Programmer's Guide

LIST Statement Label References

General Form:

```
     LIST[title][S] =[*] [([low-line][,[high-line]])]
                [F]
                         [name-1][,[name-2]]
```

Where:

```
title      = an optional descriptive title; must be a literal-
             string.

S          = specifies that a page break be performed.

F          = specifies that no page break be performed.

*          = list program statement as opposed to just line-num-
             ber.

low-line   = lowest line-number for which to show references.

high-line  = highest line-number for which to show references.

name-1     = low range of statement-label to be displayed.

name-2      = high range of statement-label to be displayed.
```

Discussion:

The LIST statement label references command produces a listing of all statement labels referenced by the program in the current LIST module, and on which program lines they are referenced.

The LIST function refers to program text in the current list module. This is set to the currently executing module whenever a program HALTs or continues, or when changed using the MODULE command, and can be referenced using LIST DT.

LIST Statement Label References (cont.)

The default format for the LIST statement label references command lists line-numbers where the specified statement label reference(s) appears. Specifying the "*" parameter causes the program statement containing the specified statement label reference(s) to be listed, in addition to the line number.

Refer to LIST general parameters for details on general parameters for all LIST statements.

The optional low/high range parameters are used to specify the range of lines accessed from which references are displayed. These operate as follows:

- If only low-line is specified, only that specific program line is accessed.
- If low-line, (comma) is specified, all program lines starting at low-line are accessed in ascending sequence.
- If ,(comma)high-line is specified, all program lines starting at the lowest ASCII sequence up to and including high-line are accessed.
- If low-line, high-line is specified, all program lines within the range of low-line to high-line inclusive are accessed.
- If no line-numbers are specified, the entire program in the current LIST module is accessed.

The optional name range parameters operate as follows:

- If only name-1 is specified, LIST statement label output for only that specific label is generated.

LIST Statement Label

- If name-1, (comma) is specified, LIST statement label output for labels starting with name-1 in ascending ASCII sequence is generated References (cont.)

```
:LIST = (100,200)
:LIST = (2000,)Boats,Trucks
:LIST = * (,4000),Hats
```

- If , (comma)name-2 is specified, LIST statement label output for labels starting at the lowest ASCII sequence up to and including name-2 is generated.
- If name-1,name-2 is specified, LIST statement label output for labels within the range of name-1 to name-2 inclusive is generated.
- If no range parameters are specified, LIST statement label output is generated for all labels referenced by the program in the current LIST module.

Examples:

```
0010 =RecordCount
   : IF A=20 THEN GOTO Exit
   : A=A+1
   : GOTO RecordCount
   : =Exit
   : B=A+10
0020 END

:LIST =
=Exit - 0010 0010
=RecordCount
       - 0010 0010

:LIST = *
=Exit-----------------------------------------------------------
0010 : IF A=20 THEN GOTO Exit
0010 :::: =Exit

=RecordCount----------------------------------------------------
0010 =RecordCount
0010 ::: GOTO RecordCount
```

LIST Statement Label References (cont.)

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

Inspection and Modification of Program Text - Section 6.5 of the Programmer's Guide

LIST STACK

General Form:

```
LIST[title] [S] STACK
```

Where:

```
title = an optional descriptive title; must be a literal-string.

S     = specifies that a page break be performed.

F     = specifies that no page break be performed.
```

Discussion:

The LIST STACK command produces a listing of all currently active FOR/NEXT loops, GOSUBs and function calls.

The information provided about the status of active subroutines includes:

- The module, line number and statement which called the subroutine

If the subroutine called is a PROCEDURE or FUNCTION, the name of the called procedure or function follows the statement.

The information provided about the status of active functions includes:

- The module, line number and statement which called the subroutine

If the subroutine called is a PROCEDURE or FUNCTION, the name of the called procedure or function follows the statement.

LIST STACK (cont.)

The information provided about the status of FOR/NEXT loops includes:

- The line-number in which the loop was initially invoked.
- The name of the index-variable.
- The current value of the index-variable.
- The exit value of the loop is displayed.
- The STEP value of the loop is displayed.

The information provided about the status of active subroutines includes:

- The line-number and statement which called the subroutine.

LIST STACK has also been updated to show the immediate commands that called any halted functions or initiated pending FOR loops. These will appear as:

```
(Halted after)  ::STOP - statement at which CONTINUE will execute
/IMM PRINT 'Funct(X)   - immediate mode statement in progress
'Funct                 - if procedure/function call, name of function
                         called
```

LIST STACK output is listed in stack order. That is, the FOR/NEXT loop or GOSUB that was executed first is listed first. The most recent entry in the stack is listed last.

The LIST STACK command can be used to determine how program execution advanced to a particular point in a program. LIST STACK also displays either where execution of a CONTINUE RETURN statement would complete or the number of iterations remaining in a FOR/NEXT loop.

Refer to LIST general parameters section for details on parameters common to all LIST statements.

LIST STACK (cont.)

LIST STACK performs no operation on the non-interpretive form of the RunTime Program.

Examples:

```
:
:LIST
0010 ;START
0030 INCLUDE T "rivfact"
0040 INPUT X
0050 PRINT 'factorial(X)
:
:MODULE "rivfact"
:
:list
0010 ; rivfact
0020 FUNCTION 'factorial(Value)/PUBLIC : ; Declare 'Factorial as Pub-
lic Function
0030 DIM Answer  : ; This is a recursive variable
0040 IF Value<0    : ; test the trivial case
   : RETURN (0)
   : END IF
0050 IF Value<=1     : l test the second trivial case
   :  RETURN(1)
   : END IF
0060 Answer=Value*'factorial(Value-1)  : ; Recursively calling itself
0065 STOP
0070 RETURN (Answer)
0080 END FUNCTION
:
```

Executing the LIST STACK command when Immediate Mode is invoked by the STOP statement in line 65 of the "rivfact" module (just prior to returning to the calling module) would yield the following output.

```
:list stack
0050 PRINT 'factorial(X)
'factorial
"rivfact"0060 Answer=Value*'factorial(Value-1)
'factorial
:

0010 FOR I=1 TO 10
       : LINPUT -A$
       : GOSUB '100
   : NEXT I
0020 DEFFN'100
   : PRINT A$
   : GOSUB '110
   : RETURN
0030 DEFFN'110
   : STOP #
   : RETURN
```

LIST STACK (cont.)

Executing the LIST STACK command when Immediate Mode is invoked by the STOP statement in line 0030 (on the first pass-through the loop) would yield the following output:

```
:LIST STACK
0010 FOR I=1 TO 10
         I=1, TO 10, STEP 1
0010 :: GOSUB '100
0020 :: GOSUB '110
```

Compatibility Issues:

This statement is supported only with Release 2.00 or greater.

This statement is not valid in Wang 2200 Basic-2.

References:

Inspection and Modification of Program Logic - Section 6.3 of the Programmer's Guide
Inspection and Modification of Variables - Section 6.4 of the Programmer's Guide
Inspection and Modification of Environment - Section 6.6 of the Programmer's Guide

LIST STACK DIM

General Form:

```
     LIST [title] [S] STACK DIM [*] [var1][,[var2]]
                  [F]
```

Where:

```
     *    = causes the contents of the specified variable range to be
            displayed.

     var1 = low variable in range to be displayed.

     var2 = high variable in range to be displayed.
```

Discussion:

The LIST STACK DIM command produces a list of variables currently defined in memory and in scope at time of execution of LIST STACK DIM within the specified range in stack order (order of definition).

The information displayed is:

- An indicator of each variables status as common (COM) or non-common (DIM).
- Current array dimension (if array-variable).
- Element length (if alpha-variable).
- Optionally, by specifying the [*] parameter, the element value is displayed:

  If numeric-variable, the numeric value is displayed.

  If an alpha-variable, the string value is displayed in both ASCII (in quotes) and HEX() representation. If a string value is longer than 16 bytes, the value is displayed on multiple lines, with the starting STR() index of each part at the beginning of the line. Non-displayable HEX codes which do not have printable character representations are displayed in string value as ".".

LIST STACK DIM (cont.)

If the variable is a FIELD identifier, the values of the #FIELDSTART, #FIELDLENGTH and $FIELDFORMAT functions are displayed. If the variable is a RECORD identifier, the value of the #RECORDLENGTH function is displayed.

Variables displayed are those which can be declared from the current context (executing module and function). Recursive variables are preceded by a RECURSIVE keyword, function private variables by a FUNCTION /STATIC.

The LIST STACK DIM output is listed in stack order. This can be very useful when diagnosing variable dimension problems (i.e., common verses non-common variables, COM and COM CLEAR statements, etc.).

Refer to LIST general parameters section for details on parameters common to all LIST statements.

The optional range parameters operate as follows:

- If only var1 is specified, LIST STACK DIM output for only that specific variable is generated.
- If var1, (comma) is specified, LIST STACK DIM output for variables starting with var1 in stack order is generated.
- If ,(comma)var2 is specified, LIST STACK DIM output for variables starting at the beginning of the stack up to and including var2 is generated.
- If var1,var2 is specified, LIST STACK DIM output for variables within the range of var1 to var2 inclusive in the stack is generated.
- If no variable range parameters are specified, LIST STACK DIM output is generated for all variables currently defined in memory in stack order.

If exactly 1 type of variable (numeric-scalar, alpha-scalar, numeric-array, alpha-array) is specified in var1 and var2, only variables of that type are listed.

LIST STACK DIM (cont.)

Array variables are specified in a LIST STACK DIM statement using a special syntax. The array designator is specified followed by an open parenthesis "(". For example, the arrays S$() and N() would be specified by:

```
0010 LIST STACK DIM S$(, N(
```

The primary difference between LIST V and LIST STACK DIM is that LIST V shows only variables declared by the program currently in memory. LIST STACK DIM displays all variables in memory in stack order even if not declared by the current program.

LIST STACK DIM performs no operation on the non-interpretive form of the RunTime Program.

Examples:

The following is an example of valid syntax:

```
:LIST STACK DIM
:LIST STACK DIM *
:LIST STACK DIM * A$(,F$
:LIST STACK DIM I$
:LIST STACK DIM * J$,
```

The following is a practical example of statement use:

```
0005 ;RIVEXP02 - Example of simple function
0010 FUNCTION 'myfunc(/POINTER mynum, myvar$32)
0020 myvar$="abcd"
0030 mynum=24
0035 STOP
0040 RETURN (1)
0050 END FUNCTION
1000 A$="xyz": A=87
1010 IF 'myfunc(A,A$)=1 THEN PRINT A,A$: ELSE PRINT "error"
```

List STACK DIM executed while the STOP of line 35 produces:

```
:LIST STACK DIM
/POINTER mynum=A
DIM /RECURSIVE myvar$32
DIM A
DIM A$16
:

:0010 COM C$16
:0020 DIM A(10),B$(10)16
:0030 FOR I=1 TO 10
    :    A(I)=I
    :    B$(I)="ABC"
    : NEXT I
:0040 C$="Test of LIST DIM"
:RUN
:LIST STACK DIM
DIM I
```

LIST STACK DIM (cont.)

```
DIM B$(10)16
DIM A(10)
COM C$16
```

Compatibility Issues:

This statement is supported only with Release 2.00 or greater.

This statement is not valid in Wang 2200 Basic-2.

References:

Inspection and Modification of Program Logic - Section 6.3 of the Programmer's Guide
Inspection and Modification of Variables - Section 6.4 of the Programmer's Guide
Inspection and Modification of Environment - Section 6.6 of the Programmer's Guide
LIST V
LIST DIM

LIST T

General Form:

```
LIST[title][S] T[*] [([low-line][,[high-line]])]
           [F]
                        {literal-string} [,{literal-string}] ...
                        {alpha-variable} [,{alpha-variable}]
```

Where:

```
S         = specifies that a page break be performed.

F         = specifies that no page break be performed.

*         = list program line as opposed to just line-number.

low-line  = lowest line-number for which to show references.

high-line = highest line-number for which to show references.
```

Discussion:

The LIST T command produces a cross-reference listing of all occurrences of one or more specific text strings in the current LIST module.

The LIST function refers to program text in the current list module. This is set to the currently executing module whenever a program HALTs or continues, or when changed with the MODULE command, and can be declared using LIST DT.

The default format for the LIST T command lists line-numbers where the specified text string appears. Specifying the "*" parameter displays the statement on the line where the specified text string was declared.

LIST T is useful when removing or replacing all occurrences of a given string of text. It can also be helpful to locate any statements within a program, such as the STOP statement.

LIST T (cont.)

NOTE: In specifying NPL keywords, case may be upper or lower. However, when searching for literals, proper case must be used. Blank spaces are ignored when entered in the search string (i.e., the search string "REM this " ignores the spaces and find all occurrences of "REMthis" as well as "REM   this" in the program).

Refer to LIST general parameters for details on general parameters for all LIST statements.

The optional low/high range parameters are used to specify the range of lines accessed from which references are displayed. These operate as follows:

- If only low-line is specified, only that specific program line is accessed.
- If low-line, (comma) is specified, all program lines starting at low-line are accessed in ascending sequence.
- If ,(comma)high-line is specified, all program lines starting at the lowest ASCII sequence up to and including high-line are accessed.
- If low-line,high-line is specified, all program lines within the range of low-line to high-line inclusive are accessed.
- If no line-numbers are specified, the entire program in the current list module is accessed.

LIST T performs no operation on the non-interpretive form of the RunTime Program.

LIST T (cont.)

Examples:

```
:LIST T(100,200)"MIN"
:LIST T(2000,)"ABC"
:LIST T*(,4000)"MIN"
:LIST T"ABC"

2000 REM
   : REM     SAMPLE PROGRAM
   : REM
2010 GOSUB '100    : REM open data file
2020 GOSUB '101    : REM read a record
2030 IF END THEN 2100             : REM quit if end of file
2040    IF F$="X" THEN R1=R1+1
   :              ELSE R2=R2+1    : REM update R1 or R1 rcd
                                     counter
2050    A=MIN(A,F2,F9*2)          : REM compute min of fields F2,F9
2060    B=MAX(B,F3,F8)            : REM compute max of F3 and F8
2070    F2,F3,F8,F9=0             : REM reset data values
2080    GOSUB '102 : REM update data record
2090    GOTO 2020  : REM iterate until eof
2100 GOSUB '103    : REM close file
2105 PRINT "MIN OF F2,F9 IS",A    : REM display results
2110 PRINT "MAX OF F3,F8 IS",B
2120 STOP
2130 DEFFN'100 : RETURN           : REM This subr opens a file
2140 DEFFN'101 : RETURN           : REM this subr reads a record
2150 DEFFN'102 : RETURN           : REM this subr writes a record
2160 DEFFN'103 : RETURN           : REM this subr closes a file

:LIST T "MIN"
"MIN"
          - 2050 2150

LIST T*"MIN"
"MIN"
2050    A=MIN(A,F2,F9*2)          : REM compute min of fields F2,F9
2105 PRINT "MIN OF F2,F9 IS",A    : REM display results

LIST T (2000,2099)"MIN"
"MIN"
      - 2050
```

Compatibility Issues:

The "*" parameter is not valid in Wang 2200 Basic-2.

LIST T is supported on NPL Revisions 2.00 and greater.

Low-line,high-line ranges are supported only on NPL Revision 3.0 or greater and are not supported on the Wang 2200.

In Wang 2200 Basic-2, if a text string appears more than once in a program line, only 1 reference for the program line appears in the LIST T output. In NPL, the reference is made for each occurrence of the text string in the program line.

LIST T (cont.)

References:

Inspection of Program Text - Section 6.5 of the Programmer’s Guide

LIST V

General Form:

```
LIST[title][S] V[*] [([low-line][,[high-line]])]
           [F]
                         [variable-1][,[variable-2]]
```

Where:

title	= an optional descriptive title; must be a literal-string.
S	= specifies that a page break be performed.
F	= specifies that no page break be performed.
*	= list program line as opposed to just line-number.
variable-1	= low range of variable to be displayed.
variable-2	= high range of variable to be displayed.
low-line	= lowest line-number for which to show references.
high-line	= highest line-number for which to show references.

Discussion:

The LIST V command produces a listing of all variables referenced by the program currently in memory, and on which program lines they are referenced.

The LIST function refers to program text in the current list module. This is set to the currently executing module whenever a program HALTs or continues, or when changed using the MODULE command, and can be referenced using LIST DT.

LIST V (cont.)

The default format for the LIST V command lists line-numbers where the specified variable(s) appears. Specifying the "*" parameter displays the statement on the line(s) where the specified variable(s) are referenced. In addition, a number of colons (":") precede the statement to indicate how many statements precede the referenced statement.

Refer to LIST general parameters for details on general parameters for all LIST statements.

The optional variable range parameters operate as follows:

- If only variable-1 is specified, LIST V output for only that specific variable is generated.
- If variable-1, (comma) is specified, LIST V output for variables starting with variable-1 in ascending ASCII sequence is generated.
- If , (comma)variable-2 is specified, LIST V output for variables starting at the lowest ASCII sequence up to and including variable-2 is generated.
- If variable-1,variable-2 is specified, LIST V output for variables within the range of variable-1 to variable-2 inclusive is generated.
- If no range parameters are specified, LIST V output is generated for all variables referenced by the program in the current LIST module.

If exactly 1 type of variable (numeric-scalar, alpha-scalar, numeric-array, alpha-array) is specified in variable-1 and variable-2, only variables of that type are listed. If different variable types are specified, all variable types are listed.

Array variables are specified in a LIST V statement using a special syntax. The array designator is specified, followed by an open parenthesis "(". For example, the arrays S$() and N() would be specified by:

LIST V (cont.)

```
0010 LIST V S$(, N(
```

The optional low/high range parameters are used to specify the range of lines accessed from which references are displayed. These operate as follows:

- If only low-line is specified, only that specific program line is accessed.
- If low-line, (comma) is specified, all program lines starting at low-line are accessed in ascending sequence.
- If ,(comma)high-line is specified, all program lines starting at the lowest ASCII sequence up to and including high-line are accessed.
- If low-line,high-line is specified, all program lines within the range of low-line to high-line inclusive are accessed.
- If no line-numbers are specified, the entire LIST module is accessed.

The primary difference between LIST V and LIST STACK DIM is that LIST V shows only variables referenced in the program currently in memory. LIST STACK DIM displays all variables in memory in stack order, even if not referenced by the current program.

LIST V performs no operation on the non-interpretive form of the RunTime Program.

Examples:

```
:LIST V
:LIST V C,L$
:LIST V*C,L$
:LIST VA,
:LIST V*,L$
:LIST V* (100,200),L$

2000 REM
   : REM     SAMPLE PROGRAM
   : REM
2010 GOSUB '100    : REM open data file
2020 GOSUB '101    : REM read a record
2030 IF END THEN 2100              : REM quit if end of file
2040 IF F$="X" THEN R1=R1+1
     :  ELSE R2=R2+1 : REM update R1 or R2 record counter
```

LIST V (cont.)

```
2050    A=MIN(A,F2,F9*2)          : REM compute min of fields F2,F9
2060    B=MAX(B,F3,F8)            : REM compute max of F3 and F8
2070    F2,F3,F8,F9=0             : REM reset data values
2080    GOSUB '102 : REM update data record
2090    GOTO 2020  : REM iterate until eof
2100    GOSUB '103 : REM close file
2105     PRINT "MIN OF F2,F9 IS",A               : REM display results
2110    PRINT "MAX OF F3,F8 IS",B
2120    STOP
2130    DEFFN'100 : RETURN        : REM This subr opens a file
2140    DEFFN'101 : RETURN        : REM this subr reads a record
2150    DEFFN'102 : RETURN        : REM this subr writes a record
2160    DEFFN'103 : RETURN        : REM this subr closes a file

:LIST V
 A    - 2050 2050 2105
 B    - 2060 2060 2110
 F$   - 2040
 F2   - 2050 2070
 F3   - 2060 2070
 F8   - 2060 2070
 F9   - 2050 2070
 R1   - 2040 2040
 R2   - 2040 2040
:LIST V(2050,2060)
 A    - 2050 2050
 B    - 2060 2060
 F2   - 2050
 F3   - 2060
 F8   - 2060
 F9   - 2050
:LIST V(2050,2060)F2,F8
 F2   - 2050
 F3   - 2060
 F8   - 2060

0005    REM Example of simple function
0010    FUNCTION 'myfunc(/POINTER mynum, myvar$32)
0020    myvar$="abcd"
0030    mynum=24
0035    STOP
0040    RETURN (1)
0050    END FUNCTION
1000    A$="xyz": A=87
1010    IF 'myfunc(A,A$)=1 THEN PRINT A,A$: ELSE PRINT "error"
:
:LIST V
 A     - 1000 1010 1010
 A$    - 1000 1010 1010
 mynum - 0010 0030
 myvar$- 0010 0020
:
:
:
:
:
```

LIST V (cont.)

Compatibility Issues:

The "*" parameter is not valid in Wang 2200 Basic-2.

LIST V is supported on NPL Revisions 2.00 and greater.

In Wang 2200 Basic-2, if a range of variables is specified (both start and end of range), both variables must be the same type.

In Wang 2200 Basic-2, program syntax must be valid to execute the LIST V command.

In Wang 2200 Basic-2, if a variable is referenced more than once in a program line, only 1 reference for the program line appears in the LIST # output. In NPL, a reference is made for each reference of the variable in the program line.

Low-line,high-line ranges are supported only on NPL Revision 3.0 or greater and are not supported on the Wang 2200.

Prior to NPL Release IV, the "*" option would display all statements on the line containing the reference.

References:

Inspection and Modification of Program Text - Section 6.5 of the Programmer's Guide

LOAD Command

General Form:

```
LOAD T [file-number,  ] {file-name}
       [disk-address, ]
       [<address-var>,]
```

Where:

```
file-name = name of program to be loaded into memory.
```

Discussion:

The LOAD command is used to load cataloged program(s) into memory. Using the LOAD command, programs can be merged or appended with programs already existing in memory. The LOAD command may also be used, after a CLEAR statement, to load a new program alone in memory. An error is generated (ERR D82 - File not in Catalog) if the program being loaded is not currently an active program file in the specified diskimage.

Examples:

```
:LOAD T/D32,"2CCOPY"
:LOAD T<A$>,"2CCOPY"
:LOAD T "SP START"
:LOAD T#1,"TESTLOG"
:LOAD T B$
:LOAD T/310,"2CBCKP"
```

Compatibility Issues:

Due to the fact that NPL executable programs are stored in an object code format, programs cannot be loaded and executed in Wang 2200 format (source).

Use of the address-var parameter is supported only on NPL Revision 3.0 or greater and is not supported on the Wang 2200.

In NPL Revision 4.0, the LOAD command acts upon the current LIST MODULE.

References:

LOAD DC Statement
Catalog Access - Section 7.3.8 of the Programmer's Guide
Loading Programs - Section 5.3 of the Programmer's Guide

LOAD Statement

General Form:

```
LOAD T [file-number,  ] {file-name                 } [line1]
       [disk-address, ] {<expression>alpha-variable}
       [<address-var>,]

      T                       [,line2][BEG line3]
```

Where:

file-name	= name of program to be loaded into memory.
expression	= number of files to be loaded from disk.
alpha-variable	= a common variable which contains the names of the programs to be loaded, trailing spaces should be included (if needed).
line1	= the line-number of the first line to be deleted from the program currently in memory. After loading, program continues from this line.
line2	= the line-number of the last line to be deleted from the program currently in memory (before loading program).
line3	= the line-number of the program where execution is to begin after loading program.

Discussion:

The LOAD statement is used to load a program or programs into memory and immediately execute the program(s).

LOAD Statement (cont.)

Line1 and line2 can specify a range of lines currently in memory that are to be deleted before the load takes place. If only line1 is specified, all program text from line1 to the last line-number in memory is deleted. If neither line1 nor line2 is specified, all program text currently in memory is deleted before the load.

Line3 specifies the starting line for program execution after the load has taken place. If not specified, program execution begins at line1, if specified, or the first line-number if line1 not specified.

The parameter "< expression>" is used to load more than one program. The numeric-expression specifies how many programs are to be loaded while the alpha-variable contains the names of the programs. Each program name in the alpha-variable must use exactly eight bytes.

Clearing Multiple Line Number Ranges on LOAD

NPL allows for multiple line number ranges to be cleared during a single load statement. The general form of the load statement has been modified for allow multiple, optional CLEAR P clauses. The new LOAD' statement also supports this syntax.

In a LOAD T< > (multiple program load), the variable used to specify the names of file(s) loaded must be a common variable defined in some module. Local variables (recursive or static) and PUBLIC variables of any type are not allowed.

When present, these optional CLEAR P clauses must follow the BEG clause (if used).

Examples:
```
    10 LOAD T#1,"PROG1" 1000,2000 BEG 10 CLEAR P 2100,200 CLEAR P 3400-3500
    10 LOAD T/D11,"PROG1"1000,2000 CLEAR P 2100,2200
    10 LOAD T<3> 1000,2000 BEG 10 CLEAR P 3400,3500 CLEAR P 10320,11000
```

The CLEAR P clause(s) are executed before any new program text is loaded from disk.

NOTE: Use of line number ranges only affects the workspace of the currently loaded module.

The numeric expression of a multiple-program load may not start with a numeric field expression. Use of alpha field expressions is allowed.

LOAD Statement (cont.)

If the standard range of line numbers to clear (LOAD T"XXX" line1,line2) is not specified, the default is to clear all program text. Therefore, CLEAR P cannot be used to replace the standard line1, line2 range, but must be used in addition to it.

For example:

10 LOAD T#1,"PROG1" CLEAR P 1000,2000 CLEAR P 2100,2200

is syntactically valid but actually clears all program text since no line1, line2 range is specified.

Examples:

```
0010 LOAD T "SP LOAD"
0010 LOAD T#1,"AR EOD 1" 5000,5999 BEG 8000
0010 LOAD T#Q,"SP MENU" 5000 BEG 8000
0010 LOAD T/D10,Q$100,8000 BEG 10
0010 LOAD T<A$>,Q$100,8000 BEG 10
0010 LOAD T QQ$
0010 LOAD T QQ$ BEG 8000
```

Compatibility Issues:

Due to the fact that NPL executable programs are stored in an object code format, programs cannot be loaded and executed in Wang 2200 format (source).

Use of the address-var parameter is supported only on NPL Revision 3.0 or greater and is not supported on the Wang 2200.

References:

LOAD Command
Catalog Access - Section 7.3.8 of the Programmer’s Guide
LOAD’

LOAD’

General Form:

```
     LOAD T <expression> 'alpha-variable [line1][,line2][BEG line3]
```

where:

expression	=	the number of programs to be loaded from disk.
alpha-variable	=	a common variable which contains the disk addresses and names of each program to be loaded.

Discussion:

The LOAD’ statement may be used to specify an explicit disk address with each program name specified as part of a multi-program load statement.

Each disk address/program name combination in the alpha-variable must be exactly 11 bytes in length with the first three bytes containing the disk address. Using file numbers or variables for the disk address is not permitted. However, $SELECT can be used to generate a standard disk address from a file number.

NOTE: **Use of line number ranges only affects the workspace of the currently loaded module.**

Multiple line number ranges can be cleared during a single load statement. The general form of the LOAD’ statement allows multiple, optional CLEAR P clauses.

When present, these optional CLEAR P clauses must follow the BEG clause (if used).

For example:

```
    10 LOAD T#1, "PROG1" 1000,2000 BEG 10 CLEAR P 2100,2000 CLEAR P3400-
    3500
    10 LOAD T/D11, "PROG1" 1000,2000 CLEAR P 2100,2200
    10 LOAD T<3> 'A$ 1000,2000 BEG 10 CLEAR P 3400,3500 CLEAR P 10320,11000
```

The CLEAR P clause(s) are executed before any new program text is loaded from disk.

LOAD’ (cont.)

NOTE: If the standard range of line numbers to clear (LOAD T"XXX" line1, line2) is not specified, the default is to clear all program text. Therefore, CLEAR P cannot be used to replace the standard line 1, line 2 range, but must be used in addition to it.

For example:

```
10 LOAD T#1,"PROG1" CLEAR P 100,200 CLEAR P 2100,2200
```

is syntactically valid but actually clears all program text since no line1, line2 range is specified.

Examples:

```
10 COM A$(3)11
20 A$(1)="D11"&"PROGRAM1"
30 A$(2)="D12"&"PROGRAM2"
40 A$(3)="$SELECT(#2)&"PROGRAM3"
50 LOAD T <3>’A$()
```

NOTE: A device address may be specified as part of the general form, but it is ignored. For example:

```
10 LOADT/D11, <3>’A$()
```

The device address D11 is ignored.

Compatibility Issues:

References:

LOAD BOOT Command

General Form:

```
LOAD BOOT [progname]
```

Where:

```
progname = an alpha-variable or literal string.
```

Discussion:

The LOAD BOOT command is used to load bootstrap programs from the native file system. A bootstrap program is a NPL program which is saved as a native operating system file and is automatically loaded and executed by the RunTime at initial start up. Refer to the appropriate NPL Supplement for details.

When specified, progname contains the native operating system file specification used to locate the bootstrap file.

When progname is omitted or blank, the program loaded is the last progname specified by a LOAD BOOT or SAVE BOOT command. Initially, the "default" boot program name is either BOOT, or the name of the boot program specified in the command line when the RunTime Program was invoked.

If the native operating system allows extensions, a .OBJ extension is assumed if no extension is specified.

The "default" boot program name is changed any time a LOAD BOOT or SAVE BOOT command is entered with an explicit filename.

The LOAD BOOT command is a programmable statement.

NOTE: Partial program loading options (line-number ranges) are not supported by the LOAD BOOT command.

LOAD BOOT Command (cont.)

LOAD BOOT and SAVE BOOT commands may also be used to inspect and replace pre-boot programs. Refer to the discussion of the /P option in the RUNTIME options section of the hardware supplement.

Examples:

Assuming a UNIX or MS-DOS based operating system:

```
:LOAD BOOT
:LOAD BOOT "UTILITY"          :REM Loads a program named "UTILITY.OBJ"
                               from the currently selected native file
                               system directory.
```

Compatibility Issues:

The LOAD BOOT command is implemented in Revision 2.00 and greater of NPL.

The LOAD BOOT command is not a valid instruction in Wang 2200 Basic-2.

In NPL Revision 4.0, the LOAD command acts upon the current LIST MODULE.

References:

SAVE BOOT

LOAD DA Command

General Form:

```
     LOAD DA T [file-number,   ] (expr1[,return-value])
               [device-address,]
               [<address-var>, ]
```

Where:

```
     expr1        = an alpha-variable or numeric-expression.

     return-value = an alpha-variable or numeric-receiver.
```

NOTE: The use of this statement is not recommended. Refer to the LOAD command as a better alternative.

Discussion:

The LOAD DA command is used to load a program into memory without accessing the catalog index. The absolute sector-number in the diskimage of the program's header record must be specified (expr1). If expr1 is an alpha-variable, the binary value of the first two bytes is used.

Use of an alpha-variable to contain sector addresses results in improper sectors being accessed if extended (greater than 16 MB) diskimages are in use and the sector numbers being accessed are greater than 65355. Refer to Section 7.3.10 of the Programmer's Guide for further programming considerations for use of extended diskimages.

The LOAD DA command is used in Immediate Mode only, and is distinguished from the LOAD DA statement which is used in program mode. The operational characteristics of the two forms of this instruction are different.

Using the LOAD DA command, programs can be merged from disk with programs already existing in memory. A warning message (Warning: Programs merged.) is generated when two or more programs are merged using the LOAD DA command. If merged programs have identical line-numbers, the original program lines in memory are overwritten by the newly loaded program lines.

LOAD DA Command (cont.)

The LOAD DA command can also be used after a CLEAR statement to load a new program alone in memory.

The return-value performs no operation in NPL. The return-value does not affect operation of the SAVE DA statement at run time. No value is returned to the return-value, if specified.

LOAD DA is a direct access instruction as opposed to a catalog instruction. That is, the Internal Device Table is not affected by a LOAD DA instruction.

Examples:

```
:LOAD DA T (100)
:LOAD DA T (100,Q)
:LOAD DA T (Q,Q)
:LOAD DA T#1, (200,Q$)
:LOAD DA T/D31, (Q$,Q$)
:LOAD DA T<A$>, (Q$,Q$)
:LOAD DA T#Q, (500+Q-R)
```

Compatibility Issues:

Due to the fact that NPL executable programs are stored in an object code format, programs cannot be loaded and executed in Wang 2200 format (source).

In Wang 2200 Basic-2, the return-value returns the sector immediately following the last sector accessed by the LOAD DA operation. The return-value does not affect operation of the SAVE DA statement in NPL. No value is returned in the return-value, if specified. The syntax is supported for compatibility purposes only.

Use of the address-var parameter is supported only on NPL Revision 3.0 or greater and is not supported on the Wang 2200.

In NPL Revision 4.0, the LOAD command acts upon the current LIST MODULE.LOAD

References:

LOAD DA Statement
Direct Access - Section 7.3.9 of the Programmer’s Guide
Extended Diskimages - Section 7.3.10 of the Programmer’s Guide

LOAD DA Statement

General Form:

```
LOAD DA T [file-number,   ] (expr1[,return-value])
          [device-address,]
          [address-var,   ]

                                [line1][,[line2]][BEG line3]
```

Where:

```
expr1        = a numeric-expression or alpha-variable specifying
               the starting sector address of the program to be
               loaded.

return-value = an alpha-variable or numeric-receiver.

line1        = the line-number of the first line to be deleted
               from the program currently in memory.  After load-
               ing, program continues from this line.

line2        = the line-number of the last line to be deleted
               from the program currently in memory (before load-
               ing program).

line3        = the line-number of the program where execution is
               to begin after loading program.
```

NOTE: The use of this statement is not recommended. Refer to the LOAD statement as a better alternative.

Discussion:

The LOAD DA statement is used to load a program into memory without accessing the catalog index. The absolute sector-number in the diskimage of the program's header record must be specified (expr1). If expr1 is an alpha-variable, the binary value of the first two bytes is used.

LOAD DA Statement (cont.)

Use of an alpha-variable to contain sector addresses results in improper sectors being accessed if extended (greater than 16 MB) diskimages are in use and the sector numbers being accessed are greater than 65355. Refer to Section 7.3.10 of the Programmer's Guide for further programming considerations for use of extended diskimages.

Line1 and line2 can specify a range of lines currently in memory that are to be deleted before the load takes place. If only line1 is specified, all program text from line1 to the last line-number in memory is deleted. If neither line1 nor line2 is specified, all program text currently in memory is deleted before the load.

Line3 specifies the starting line for program execution after the load has taken place. If not specified, program execution begins at line1, if specified, or the first line-number if line1 not specified.

The return-value performs no operation in NPL. The return-value does not affect operation of the SAVE DA statement at runtime. No value is returned to the return-value, if specified.

LOAD DA is a direct access instruction as opposed to a catalog instruction. That is, the Internal Device Table is not affected by a LOAD DA instruction

Examples:

```
0010 LOAD DA T (100)
0010 LOAD DA T (100,Q$)  5000,5999 BEG 8000
0010 LOAD DA T (Q$,Q$)  5000
0010 LOAD DA T/D10,(15326)  8000
0010 LOAD DA T<A$>,(15326)  8000
0010 LOAD DA T#Q,(1000)  8000,8100
```

Compatibility Issues:

Due to the fact that NPL executable programs are stored in an object code format, programs cannot be loaded and executed in Wang 2200 format (source).

LOAD DA Statement (cont.)

In Wang 2200 Basic-2, the return-value returns the sector immediately following the last sector accessed by the LOAD DA operation. The return-value does not affect operation of the LOAD DA statement in NPL. No value is returned in the return-value, if specified. The syntax is supported for compatibility purposes only.

Use of the address-var parameter is supported only on NPL Revision 3.0 or greater and is not supported on the Wang 2200.

References:

LOAD DA Command
Direct Access - Section 7.3.9 of the Programmer's Guide
Extended Diskimages - Section 7.3.10 of the Programmer's Guide

LOAD RUN

General Form:

```
LOAD RUN [T] [file-number,   ] [prog-name]
             [device-address,]
             [address-var,   ]
```

Where:

```
prog-name = a literal-string or alpha-variable specifying the
            name of the program to be run.  The default program
            name is "START".
```

NOTE: The use of this statement is not recommended. Use program modules as a better alternative.

Discussion:

LOAD RUN is used to clear memory and load and execute a program. The program name is specified as a literal-string or as an alpha-variable. Before the program is loaded, all program text and variables are removed from memory. After the program is loaded into memory, execution begins at the first line-number of the program.

If the program name is not specified, the default program name "START" is used.

Examples:

```
0010 LOAD RUN
0010 LOAD RUN"BEGIN"
0010 LOAD RUN T"BEGIN2"
0010 LOAD RUN T/D10,"START"
0010 LOAD RUN TA$,"START"
0010 LOAD RUN T#2,Q$
0010 LOAD RUN T#Q,"SP START"
```

Compatibility Issues:

Due to the fact that NPL executable programs are stored in an object code format, programs cannot be loaded and executed in Wang 2200 format (source).

Use of the address-var parameter is supported only on NPL Revision 3.0 or greater and is not supported on the Wang 2200.

Use of program modules is only supported in NPL Revision 4.0 or greater and is not supported on the Wang 2200.

LOAD RUN (cont.)

In NPL Revision 4.0, programs referenced with the "LOAD RUN" command are executed in the current run module.

References:

LOG Function

General Form:

```
LOG (numeric-expression)
```

Discussion:

The LOG function computes the natural logarithm of a numeric-expression. This is valid wherever a numeric-expression is legal.

Examples:

```
0010 B(3,9)=(LOG(K2)+10)/LOG(10)
0010 R7=INT(LOG(D9+9))
:0010 INPUT E
:0020 X=100+LOG(E*100)
    : PRINT X
:RUN
? 10
 106.907755278982
```

Compatibility Issues:

Due to the use of different algorithms, results of these functions may differ from functions evaluated on a Wang 2200. In general, however, the functions are accurate to 13 significant digits.

References:

LOOP

General Form:

```
LOOP
```

Discussion:

The LOOP statement allows skipping the execution of the remainder of the body of a structured loop, which may be either WHILE...WEND, REPEAT...UNTIL or FOR-BEGIN...NEXT type. When it occurs inside nested loops, only the body of the innermost loop is skipped.

When executed, control is transferred to the WEND statement of the current WHILE...WEND loop, to the UNTIL statement of the current REPEAT...UNTIL loop or to the NEXT statement of an enclosing FOR/BEGIN...NEXT loop.

Examples:

```
0010 PRINT "All the even numbers from 1 to 10"
   : cur_number = 1
   : REPEAT
   : cur_number += 1
   :   ; odd so LOOP back to the top of the REPEAT/UNTIL body
   :   IF MOD(cur_number,2) <> 0 THEN LOOP
   :   PRINT cur_number
   : UNTIL cur_number > 10

0020 num_times_LOOPed = 0
   : FOR row = 1 TO 4 BEGIN
   :      FOR column = 1 TO 4 BEGIN
   :           IF column > 2
   :            ;I will execute this IF statement twice within the FOR
                 column loop
   :             num_Times_LOOPED +=1
   :              LOOP
   :           END IF
   :      NEXT column
   : NEXT row
   : PRINT "Should HAVE looped 8 times: ";num_times_LOOPed
```

Compatibility Issues:

LOOP (cont.)

References:

FOR/BEGIN
UNTIL
NEXT
WHILE
WEND
BREAK

$MACHINE

General Form:

```
alpha-receiver = $MACHINE
```

Discussion:

$MACHINE is a 64-byte system variable containing information about the environment in which the RunTime Program is currently operating. This information may be used by a NPL application program to implement conditional logic for option selection based upon the current environment. As of Revision 4.0 of NPL, 29 bytes are returned by $MACHINE. New bytes will likely be added in future revisions.

NOTE: New values may be added for new hardware/operating system ports of NPL; refer to the NPL Supplements for details on specific $MACHINE values for the operating system.

$MACHINE may not be modified by the NPL program. Placing $MACHINE on the left side of an assignment statement results in a syntax error.

Refer to the appropriate NPL Supplement(s) for specific details of the hardware-dependent features for specific machines.

Specifically, the following information is available:

Byte 1	RunTime Version "I" for MS-DOS/Novell Netware "N" for MS-Windows "P" for Phar Lap "S" for SuperDOS "X" for Intel XENIX (286) "A" for Intel UNIX (386 models) "U" for Motorola 68000 UNIX "V" for VMS "W" for MS-DOS on Wang PC's

$MACHINE (cont.)

Byte 2	Hardware Manufacturer Code	
Byte 3	Monitor Type	
Byte 4	Graphics Enabled ("G" = truebox graphics available; " " = no truebox graphics available).	
Byte 5	Hardware Model Code. Refer to NPL Supplement for a list of valid values. On SuperDOS, indicates binary # of overflow areas set up in the memory share module.	
Byte 6	Number of NPL users in the RunTime before this task executed the RunTime. On Xenix and UNIX, indicates number of users in the Niakwa RunTime after this task was executed.	
Byte 7	RunTime type in use - "I" = interpretive version; "P" = non-interpretive version.	
Byte 8	Display width in binary. Will always be HEX(50) on 80-column screens. On screens that support more than 80 columns, will be the width currently enabled. Refer to Section 7.3.23 of the Programmer's Guide for details on enabling wider screen widths.	
Byte 9	Terminal type-refer to the NPL Supplements for possible values.	
Byte 10	Math co-processor present.	
	(00)	Indicates that no co-processor is present or that use of the co-processor is not supported on the hardware version in use.
Byte 11	HEX(00)	Standard model in use. Maximum partition size is 56K. This value is returned for NPL revisions prior to Revision 3.0 where the "S" startup option was not used or was not available.
	HEX(01)	Extended model in use. A 56k program segment and 64K variable segment are available. This value is returned by NPL revisions prior to Revision 3.0 when the "S" option is in use.
	HEX(02)	Large model in use. This value is returned by NPL Revision 3.0 or greater.
Byte 12	Number of colors available. Refer to Chapter 6 of the NPL Supplements for details on color support in NPL.	
Byte 13	Maximum number of authorized users (in binary).	
Byte 14	Reserved.	

$MACHINE (cont.)

Byte 15	Reserved.
Byte 16	Maximum number of Device Equivalence Table entries available. Equal to the number of devices specified in the "D" startup option. If the "D" option is not used, the default value of 16 DET entries is used.
Byte 17	Number of Device Equivalence Table entries currently in use. A DET entry is defined as being in use when a NPL address of file # is assigned. This value is stored in binary format. Typical use of this value would be to determine if DET entries are available for assignment.

For example:

```
0010 DIM A$50,M$64
0020 M$=$MACHINE
0030 A$=$DEVICE(/D20)    :REM Save current DET entry for D20, if any-
thing
0040 IF A$<>" " THEN 200 :REM D20 is already in use - therefore we can
                          use it with no concern about overflow as long
                          as we restore it when done.
0050 M=VAL(STR(M$,16,1)  :REM Maximum number of DET entries
0060 C=VAL(STR(M$,17,1)  :REM Current number of DET entries in use
0070 IF C<M THEN 200     :REM At least one entry is available
0080 REM Routine to handle no DET entry available condition
.
.
.
0200 $DEVICE(/D20)="MYFILE.BS2":REM Set D20 to device required
.
.
.
0300 $DEVICE(/D20)=A$    :REM Done with MYFILE.BS2, restore original DET
                          entry
```

This example demonstrates how to assign a temporary DET entry without risk of overflowing the DET table.

NOTE: Even if all DET entries are currently used, this technique still succeeds as long as the specified address is already defined in the DET. In this case, the same DET slot is reused and the original value is restored after processing of the temporary entry is complete.

$MACHINE (cont.)

Byte 18	Indicates whether or not the task in use was STARTED in a background partition.	
	HEX(00)	Indicates that the task was started in foreground.
	HEX(01)	Indicates that the task was started in background.
Byte 19	Byte 19 of $MACHINE contains the status of $DEMO keyboard redirection and keyboard logging. Possible values for this byte are:	
	HEX(01) bit = 1	Indicates that keyboard redirection from a $DEMO file is in effect. When keyboard redirection is not in effect, this bit is off. This bit is set to 0 on all conditions which terminate a $DEMO script. This includes cancellation by the operator and end of file conditions. End of file conditions are detected only AFTER a keyboard input statement is executed where there are no more keystrokes in the specified $DEMO file. Refer to $DEMO for further detail on demo scripts.
	HEX(02) bit = 1	Indicates that keyboard logging is in effect. Keyboard logging is in effect whenever the current SELECT LOG address is anything other than /000 (the nul device) and the SELECT LOG status is ON. This bit is 0 whenever SELECT LOG status is OFF or the SELECT LOG address is /000. SELECT LOG status may be set either by use of the SELECT LOG statement or by the operator.

Applications which examine this byte should use the logical AND operation to test the specific bit to be examined.

For example:

```
0010 DIM M$64,X$1
0020 M$=$MACHINE
0030 X$=STR(M$,19,1) AND HEX(01)
0040 IF X$=HEX(01) THEN PRINT "$DEMO IS IN EFFECT"
0050 X$=STR(M$,19,1) AND HEX(02)
0060 IF X$=HEX(02) THEN PRINT "KEYBOARD LOGGING IS IN EFFECT"
```

$MACHINE (cont.)

Byte 20	Indicates if the version of NPL running is a 32-bit or non-32-bit.	
	HEX(00)	Non-32 bit RunTime in use.
	HEX(01)	32-bit RunTime in use.
Byte 21	Contains the maximum number of entries (in binary) allocated to the handle table (in K) during a RunTime session.	
Byte 22	Indicates the XMS usage as specified in the /m and /u startup options.	
Byte 23	Contains the current row position of the mouse pointer when a mouse event occurs (if it is on the screen).	
Byte 24	Contains the current column position of the mouse pointer when a mouse event occurs (if it is on the screen).	
Byte 25, 26	Extended field equivalent to $MACHINE byte 6 (number of active users in the RunTime before this task booted). For systems with 256 or more users, these bytes must be used to get an accurate user count	
Byte 27, 28	Extended field equivalent to $MACHINE byte 13 (maximum number of authorized users). For systems with 256 or more users, these bytes must be used to determine the max user count.	
Byte 29	Indicates whether keyboard mouse events are supported	
	HEX(00)	Default; mouse is not available.
	HEX(01)	Mouse available. Under DOS 3.0 and greater, HEX(01) will only occur if the NPL /K startup option is specified and a mouse driver is installed and detected. Under MS-Windows, HEX(01) will occur upon detection of an installed mouse device.

Examples:

The following program sets the replacement attribute for underline on the IBM color monitor to bright white on red background:

```
0010 DIM X$32,Z$64
0020 X$=$MACHINE                    : REM STORE IN X$
0030 IF STR(X$,1,1)"I" THEN GOTO 90 : REM IF IBM VERSION
0040 REM IBM VERSION
0050 IF STR(X$,3,1)"C" THEN 90      : REM SKIP IF NOT COLOR MONITOR
0060 Z$=$OPTIONS                    : REM FETCH CURRENT OPTIONS
0070 STR(Z$,1,1)=HEX(4F)            : REM SET UNDERLINE AS WHITE ON
                                      RED
0080 $OPTIONS=Z$                    : REM IMPLEMENT
0090 REM DONE
```

$MACHINE (cont.)

Compatibility Issues:

This statement is supported only with Release 1.03 or greater.

This statement is not valid in Wang 2200 Basic-2.

Specific values for $MACHINE are detailed in the appropriate NPL Supplement.

Bytes 6-13 contain valid information only on NPL Revision 2.01 or greater.

Bytes 16-20 contain valid information only on NPL Revision 3.00 or greater.

Bytes 21-28 contain valid information on on NPL Revision 3.20 or greater.

Bytes 29 contain valid information on on NPL Revision 4.00 or greater.

References:

Chapter 9 of the Programmer's Guide.

MAT CON

General Form:

```
MAT numeric-array = CON [(dim1[,dim2])]
```

Where:

```
dim1, dim2 = numeric-expressions specifying new dimensions of
             the numeric-array.
```

Discussion:

The MAT CON statement is used to set all elements of a numeric-array to the numeric value of 1. In addition, the specified numeric-array is optionally redimensioned according to the new dimension parameters, if specified.

Examples:

```
0010 MAT Z = CON
0010 MAT X = CON(5,10)
0010 MAT P = CON(Q,S)
0010 MAT K = CON(15)
```

Compatibility Issues:

References:

MAT COPY

General Form:

```
MAT COPY[-]source-alpha-variable [<[s][,[n]]>] TO[-]
                 receiver-alpha-variable [<[s][,[n]]>]
```

Where:

```
s = numeric-expression

n = numeric-expression
```

Discussion:

The MAT COPY statement is used to copy the contents of one alpha-variable or array to another. The copy is performed character-by-character, transferring the contents of the source-alpha-variable to the receiver-alpha-variable. The copy stops when the receiver-alpha-variable is completely filled. If the receiver-alpha-variable is larger than the source-alpha-variable, blanks are used to fill the remaining characters after all characters of the source-alpha-variable have been received.

The source-alpha-variable or receiver-alpha-variable can be modified by the STRING function. The source-alpha-variable and receiver-alpha-variable can be the same variable.

Either alpha-variable may be modified by the "s" and "n" parameters, which specify that a substring of the alpha-variable is used. The "s" parameter specifies the first position to use, the "n" parameter specifies a count. These parameters are equivalent to the first and second parameters of a STR() function, and have the same default value. The syntax is supported only to maintain compatibility with Wang 2200 Basic-2.

The [-] option on the source-alpha-variable reverses the order in which the characters are transferred, beginning with the last character of source-alpha-variable and ending with the first character.

MAT COPY (cont.)

The [-] option on the receiver-alpha-variable reverses the order in which the characters are received. The first character received is stored in the last position of the receiver-alpha-variable. The second character received is placed in the next-to-last position. The process continues until all characters have been received. Blanks are inserted if the receiver-alpha-variable is larger than the source-alpha-variable.

MAT COPY is useful for inserting and deleting elements of alpha-arrays which are maintained in order.

Examples:

```
0010 MAT COPY A$() TO B$()
0010 MAT COPY -B$() TO C$()
0010 MAT COPY A$ TO B$()
0010 MAT COPY A$() TO -B$
0010 MAT COPY -STR(A$,6,7) TO -B$()
```

This example shows how to use MAT COPY to maintain an array in sorted order while adding new entries:

```
0010 DIM T$(100)32,N$32 :REM Table of names
  : N=0                 :REM Number of names
0020 INPUT "New Name",N$
  : MAT SEARCH T$() ,N*32,=STR(N$) TO L$ STEP 32
                        :REM Find the first name = to the name
                         entered
  : I=VAL(L$,2)
  : IF I=0 THEN 40      :REM I=Starting byte position
  :    L=N*32+1-I       :REM L=Number of bytes
  :    MAT COPY -T$() <I,L> TO -T$() <I+32,L>
                        :REM Move the array down
  :    GOTO 50
0040 I=N*32+1
0050 STR(T$(),I,32)=N$  :REM Insert the new name
  :  N=N+1
  : FOR I=1 TO N
  :    PRINT T$(I)
  : NEXT I
  : GOTO 20
```

MAT COPY (cont.)

```
READY (NIAKWA RUNTIME) PARTITION 01
:0010 DIM A$(3)10,B$(3)10,C$31
:0020 MAT INPUT A$
:0030 MAT COPY A$() TO B$()
:0040 MAT COPY A$() TO -C$
:0050 LIST DIM *
:RUN
? Niakwa,NPL,RunTime
DIM A$(3)10
(1)        "Niakwa    "     HEX(4E69 616B 7761 2020 2020)
(2)        "NPL  "          HEX(4E50 4C20 2020 2020 2020)
(3)        "RunTime   "     HEX(5275 6E54 696D 6520 2020)
DIM B$(3)10
(1)        "Niakwa    "     HEX(4E69 616B 7761 2020 2020)
(2)        "NPL  "          HEX(4E50 4C20 2020 2020 2020)
(3)        "RunTime   "     HEX(5275 6E54 696D 6520 2020)
DIM C$31
         "  emiTnuR     "  HEX(2020 2020 656D 6954 6E75 5220 2020 2020)
 STR(17) "LPN     awkaiN"  HEX(2020 4C50 4E20 2020 2061 776B 6169 4E)
```

Compatibility Issues:

References:

MAT IDN

General Form:

```
MAT numeric-array = IDN [(dim1,dim2)]
```

Where:

```
dim1,dim2 = numeric-expressions specifying new dimensions of
            the numeric-array.
```

Discussion:

The MAT IDN statement is used to assign the specified square matrix the form of an identity matrix.

The dim1 and dim2 parameters redimension the matrix to the specified size. The new dimension size must be equal to or smaller than the original array. The new array must also be a square array.

Examples:

```
0010 MAT A=IDN(5,5)
0010 MAT B=IDN(100,100)
0010 MAT C1=IDN
0010 MAT D=IDN(X,Y)

:0010 MAT B=IDN(4,4)
    : MAT PRINT B
:RUN
 1              0               0               0
 0              1               0               0
 0              0               1               0
 0              0               0               1
```

Compatibility Issues:

References:

MAT INPUT

General Form:

```
MAT INPUT array-variable [(dim1[,dim2]) [length]]
                  [,array-variable [(dim1[,dim2]) [length]]]...
```

Where:

```
dim1,dim2 = numeric expressions specifying new dimensions of the
            array.

length    = expression specifying the length of each element in
            an alpha-array.  Default length is 16.
```

Discussion:

The MAT INPUT statement is used to input data from the keyboard into one or more array variables.

When a MAT INPUT statement is executed, the input prompt "?" is displayed and execution is suspended until the requested data is entered. As data is entered, elements are assigned row by row until the array is filled.

More than one element may be entered at a time by separating values with the comma delimiter. Entering no data when requested (just pressing RETURN) ends the MAT INPUT operation. The contents of the remaining array elements is unchanged.

Data entered using the MAT INPUT statement must be compatible with the array type being assigned (e.g., numeric data must be entered into a numeric-array). Leading spaces or commas can be entered as alpha data by enclosing them in quotation marks.

If invalid data is entered, an error is generated and the data must be reentered starting from the element in error.

MAT INPUT (cont.)

Examples:

```
0010 MAT INPUT A$
0010 MAT INPUT C$(5,2)10
0010 MAT INPUT A(5,5),B$,C(2,3)
0010 MAT INPUT X$(10,10)32
0010 MAT INPUT J$(10,2),I

:10 DIM A(2,3),B$(2)8
:20 MAT INPUT A,B$

:RUN
```

When executed, this program prompts the operator for the entry of 8 fields. The responses to the first six fields are placed in A() and must contain valid numeric data. The remaining two responses are placed in B$() and may contain alpha data.

Compatibility Issues:

References:

MAT INV

General Form:

```
MAT p = INV(q)[,[det][,norm-det]]
```

Where:

```
p,q      = numeric-array names.

det      = a numeric-receiver which is assigned the value of the
           determinant of array p.

norm-det = a numeric-receiver which is assigned the value of the
           normalized determinant of array q.
```

Discussion:

The MAT INV statement is used to assign matrix P the inverse of matrix Q. Array P can be assigned the inverse of itself by placing it on both sides of the statement.

Array Q must be a square (n x n) matrix. If det is not specified and Q is a singular matrix, an error X72 (Singular Matrix) occurs.

NOTE: For internal reasons, MAT INV cannot invert a numeric matrix larger than (2048,2048)--a 16MB array.

MAT INV is typically used to determine the solution to a system of linear equations in n variables. Given the system:

$$\begin{aligned} q_{11}*x_1 + q_{12}*x_2 + \ldots + q_{1n}x_n &= y_1 \\ q_{21}*x_1 + q_{22}*x_2 + \ldots + q_{2n}x_n &= y_2 \\ &\vdots \\ q_{n1}*x_1 + q_{n2}*x_2 + \ldots + q_{nn}x_n &= y_n \end{aligned}$$

with all q_{ij} and y_j known and variables x_i to be determined.

In matrix notation, this set of equations may be written as QX=Y.

MAT INV (cont.)

In a given square matrix Q, the result of MAT P=INV(Q) can be used to determine a solution matrix for Q by computing MAT B=P*Q. The matrix P is always non-singular. The matrix B is upper triangular. (If Q is non-singular, P=INV(Q), and B is identity matrix.)

A row of zeros in row r of B indicates that the row r of Q is linearly dependent on other rows of Q. The elements of row r of P are the coefficients of a linear dependence of rows of Q.

In solving linear equations, if the equation is Q*X=Y where Q and Y are given, the equation has a solution if P*Y is zero on all rows where B is all zeros.

If P is singular and meets the above zero-rows criteria, the equation is solvable, but the solution is not unique. The matrix B can be used to determine a basis for the set of all solutions by solving the much simpler B*X=P*Y (which can be reduced from last row up).

If Q is non-singular, the above discussion can be summarized by the fact that the equation has the unique solution X=P*Y.

Examples:

```
10 MAT D=INV(B),X,Y
```

This example illustrates the use of MAT INV is an actual problem. Three separate items have been purchased as a group on three separate occasions:

 5 nuts, 7 bolts and 8 screws - Total cost is 83 cents.
 8 nuts, 2 bolts and 4 screws - Total cost is 52 cents.
10 nuts, 15 bolts and 9 screws - Total cost is $1.35.

What is the cost for each nut, bolt and screw?

Let x_1 be the cost of a nut.
Let x_2 be the cost of a bolt.
Let x_3 be the cost of a screw.

MAT INV (cont.)

The equations to be solved are:

$$5 x_1 + 7 x_2 + 8 x_3 = 83$$
$$8 x_1 + 2 x_2 + 4 x_3 = 52$$
$$10 x_1 + 15 x_2 + 9 x_3 = 135$$

Following the notational conventions,we use the following NPL program:

```
:0010 DIM Q(3,3),P(3,3),Y(3),X(3)
:0020 MAT READ Q
:0025 DATA 5,7,8
    : DATA 8,2,4
    : DATA 10,15,9
:0030 MAT READ Y
:0035 DATA 83
    : DATA 52
    : DATA 135
:0040 MAT P=INV(Q)
:0050 MAT X=P*Y
:0055 FOR T=1 TO 3
    :   X(T)=ROUND(X(T),6) : REM ROUND OFF TO 6 DECIMALS
    : NEXT T
:0060 MAT PRINT X
:RUN
 3
 4
 5
```

That is, a nut costs 3 cents, a bolt costs 4 cents and a screw costs 5 cents.

NOTE: Rounding of results to a reasonable number of decimals is typical when performing numerical analysis.

Compatibility Issues:

Due to the large number of calculations involved in computing a matrix inverse, the effect of the different internal numeric forms and rounding error may be especially noticeable on this instruction. The usual cautions concerning the accuracy of the result when the normalized determinant is small relative to 1 also apply.

Unlike Wang BASIC-2, where if the matrix is singular the resultant matrix P has "undefined" values, the NPL resultant may be useful to determine whether there are solutions.

References:

MAT MERGE

General Form:

```
MAT MERGE source-array[(f1[,f2])] TO
                  status-var,temp-var,pointer-array
```

Where:

```
source-array  = a two-dimensional alpha-array containing data to
                be merged.

(f1,f2)       = optional byte range (start & length) which de-
                fines a field within each source-array element:

                f1 = numeric expression which specifies the
                     starting position of the field.

                f2 = numeric expression which specifies the
                     length of the field.

status-var    = an alpha-variable used to store merge status.

temp-var      = an alpha-variable used by the system as workspace.

pointer-array = an array with elements of length two or four used
                to store subscripts of the source-array.
```

Discussion:

The MAT MERGE statement is used to merge two or more sorted data files into a specified output file, sorted in ascending order.

The sizes of variables required by MAT MERGE and the structure of these variables depends on the type of pointer-variable used. If the pointer-variable consists of two-byte elements, only small arrays (up to 255 rows, up to 254 columns) can be merged. If the pointer-variable consists of four-byte elements, large arrays (up to 65535 rows, up to 65534 columns) can be merged. The following information refers to these two cases as the small and large pointer cases.

MAT MERGE (cont.)

The source-array is an alpha-array which acts as a "buffer" for the data being merged. The source-array has one row for each input file being merged. The number of columns in the source-array is arbitrary, but should be as large as possible.

The status-variable is an alpha-variable which maintains status information about the merge operation. It must be dimensioned to a minimum size of the number of rows in the source-array plus one in the small pointer case, or double this amount in the large pointer case.

The temp-variable is an alpha-variable used as a work area by the system. The temp-variable must have at least twice as many bytes as the number of rows in the source-array in the small pointer case, or at least double this amount plus 3 bytes in the large pointer case.

The pointer-array is an alpha-array whose elements are two or four bytes in length.

Operation of MAT MERGE

MAT MERGE compares elements of each row of the source-array and produces a merged list of subscripts in the pointer-array. For each comparison, the subscripts of the lowest element are placed in the next element of the pointer-array. When an element in the source-array has been selected in this fashion, the status-variable (refer below) is updated to point to the next element in that row and that element is used for the next comparison.

Use of Field Parameters

The MAT MERGE statement allows the merge operation to be performed on a substring of the source-array elements. The substring is specified by including the (f1,f2) parameters. The f1 parameter specifies the starting position of the sort substring and f2 specifies the number of characters in the string (the default value for f2 assumes all remaining characters in the element). If field parameters are used, data stored in each row must be in sorted order based on the specified field.

MAT MERGE (cont.)

Use of the Status-Variable

As stated above, the status-variable must contain one byte for every row in the source-array plus one extra byte (the n+1 byte) or double this for the large pointer case. It is easiest to consider the status variable as a string array of (n+1) elements each with one byte for the small pointer case, or two bytes for the large pointer case.

The first (n) elements are "row" status information, and determine which element in the row should be used for the next comparison. These values must be set initially to HEX(01) for the small pointer case, or HEX(0001) for the large pointer case. The "row" elements are updated by the merge operation as described above. When all elements in a row have been used up by the merge operation, the value HEX(FF) for small pointer case or HEX(FFFF) for large pointer case is placed in the corresponding row element. As rows are replenished by the program, it is the responsibility of the program to reset the corresponding "row" element to HEX(01) for the small pointer case, or HEX(0001) for the large pointer case.

The n+1 element is used by the merge operation to provide information relating to the termination of the merge operation.

If the n+1 element of the status-variable equals HEX(00) for the small pointer case, or HEX(0000) for the large pointer case, then the merge terminated because the pointer-array is full.

A MAT MOVE operation is then required to move the merged data from the source-array to the output file. After the MAT MOVE operation, another MAT MERGE can then be performed on the remaining data in the source-array.

If the n+1 element contains any other value, then the MAT MERGE terminated with an empty row in the source-array. The value of the n+1 element (as a one-byte binary value for the small pointer case, or as a two-byte binary value for the large pointer case) indicates the empty row. The remaining status-variable elements points to the next element in each row to be merged on the next MAT MERGE execution.

MAT MERGE (cont.)

Multiple Execution of MAT MERGE

The complete merging of several large files typically requires more than one pass of the MAT MERGE statement.

The merge operation completes each time the locator-array becomes full or the end of a merge-array row is encountered. At this point, a move operation is required to move the merged data from the merge-array to the output file. The next merge operation can then be performed on the remaining data. Each time a merge operation encounters the end of a merge-array row, the operation is ended (even if data is available in subsequent rows). For this reason, it may be advisable to replenish an empty merge-array row and reset the control-variable to point to the newly entered data as indicated above.

Upon completion of a MAT MERGE execution, the application should check for one of the following:

- Did the merge terminate because the pointer array is full?
- Did the merge terminate because of an empty row in the merge-array?

This can be determined by examining the control-variable as indicated above.

Examples:

```
0010 MAT MERGE B$() TO C$(), D$(), E$()
0010 MAT MERGE B$()(3,4) TO C$,D$,E$()
```

The following sample program illustrates the use of MAT MERGE to sort three cataloged data files.

MAT MERGE (cont.)

NOTE: This example assumes that the three data files are in sorted order.

```
0010 REM Sample use of MAT MERGE statement
0020  REM Assumes 3 data files with sorted elements are to be merged.
   : REM The data records are 60 bytes long, stored 4 per logical
      record.
   : REM If last blocks are not full, unused records must be high-
      values.
   :  REM The sort key is located at byte 10 of each record and is 5
      bytes long.
0030  REM The names of the files are "FILE1","FILE2",and "FILE3" on
0040  REM devices #1,#2,and #3 respectively.
0050  REM The output goes to "OUTPUT" on device #4.
0060  DIM S$(3,4)60            : REM source array holds 1 L.R. per file
    : DIM C$(4)1               : REM status-variable
    : DIM T$(3)2               : REM temp-variable
    : DIM P$(10)2              : REM pointer size is arbitrary
0080 DIM O$(4)60               : REM used to write records
0090 GOSUB 200                 : REM initialization
0120 MAT MERGE S$()(10,5) TO C$(),T$(),P$()
   : IF P$(1)=HEX(0000) THEN 190: REM check no more to merge
   :  GOSUB 300                : REM move merged keys to output
   :  GOSUB 400                : REM refill empty row if any
   :  GOTO 120                 : REM continue merging
0190 DATA SAVE DC #4,END       : REM merge complete, finish up
   : DATA SAVE DC CLOSE ALL
   : STOP
0200 DATA LOAD DC OPEN T#1,"FILE1"
   : DATA LOAD DC OPEN T#2,"FILE2"
   : DATA LOAD DC OPEN T#3,"FILE3" : REM open input files
   : FOR X=1 TO 3
   : GOSUB ’1(X)           : REM read initial records
   : NEXT X
   : DATA LOAD DC OPEN T#4,"OUTPUT"
   : O=1                   : REM set next available output record
   : RETURN
0250 DEFFN’1(F)            : REM refill source array for file #F
   : DATA LOAD DC #F,S$(F,1),S$(F,2),S$(F,3),S$(F,4)
   : IF END THEN 260       : REM check end of file
   : C$(F)=HEX(01)         : REM set status variable to first in row
   : GOTO 290
0260 C$(F)=HEX(FF)         : REM set status variable to ’all used up’
0290 RETURN
0300 L=1
0330 D=4                   : REM set counter before move
   : MAT MOVE S$(),P$(L),D TO O$(O): REM move records to output buffer
   : L=L+D                 : REM advance pointer index by number moved
   : O=O+D                 : REM advance output index by number moved
   : IF O THEN 350       : REM do we have a full output block yet?
   :   DATA SAVE DC #4,O$() : REM yes, save it
   :   O=1                 : REM reset available output record
   :   GOTO 330            : REM repeat until all pointers are used up
0350 RETURN
```

MAT MERGE (cont.)

MAT MERGE (cont.)

```
0400 IF C$(4)=HEX(00) THEN 480   : REM are any rows used up?
   :   GOSUB '1(VAL(C$(4))) : REM yes, refill used up row
0480 RETURN
.
```

Compatibility Issues:

Use of 4-byte pointer arrays is supported in NPL Revision 4.0 or greater, and is not compatible with the Wang 2200.

References:

MAT MOVE
MAT SORT

MAT MOVE

General Form:

```
     MAT MOVE source-array [,pointer-array] [,counter-var]

                       TO {receiver-array        }
                          {receiver-array-element}
```

Where:

```
     source-array   = {alpha-array[(f1[,f2])]}
                      {numeric-array         }

     pointer-array  = {alpha-array        }
                      {alpha-array-element}

     receiver-array = {alpha-array[(f1[,f2])]}
                      {numeric-array         }

     receiver-
     array-element  = {alpha-array-element[(f1[,f2])]}
                      {numeric-array-element         }

     f1             = numeric-expression which specifies the starting
                      position of a sub-field within a source-array or
                      receiver-array or receiver-array-element.

     f2             = numeric-expression which specifies the length of
                      a sub-field within a source-array or receiver-ar-
                      ray or receiver-array-element.

     counter-var    = numeric-scalar which contains the maximum number
                      of elements to be moved when the statement is
                      executed and contains a count of the number of
                      elements actually moved when execution is com-
                      pleted.
```

MAT MOVE (cont.)

Discussion:

MAT MOVE is used to transfer data from one array to another and optionally convert data between numeric and alpha arrays. MAT MOVE is frequently used in conjunction with MAT SORT and MAT MERGE in order to process disk-based files.

General Features

MAT MOVE transfers data on an element by element basis from the source-array to the receiver-array.

The source-array is always processed starting at the first element.

Source and receiver arrays do not have to have the same number of dimensions or an equal number of elements. The system automatically handles conversion between two dimensioned matrices and single dimension matrices as well as conversion between two dimension matrices of different dimensions.

If the receiver-array-element designation is used, data is transferred into the receiver-array starting at the specified element.

The move operation terminates when either the source-array has been completely moved or the receiver-array has been filled, unless a counter-var or pointer-array are used (refer to the discussion below).

The Field Parameters

The data to be moved from alpha-source-arrays is defined by field parameters. The portion of the element to receive data in alpha-receiver-arrays is also defined by field parameters. In both cases, if the field parameters are not specified, the entire element is the field. If the size of the field in the receiver-array does not match the size of the field in the source-array, data values are truncated or extended with blanks as required. Data in elements outside of the specified field is not affected by the move operation.

Specific field sizes in alpha-source-arrays or alpha-receiver arrays may be specified by the f1,f2 parameters. If specified, f1 indicates the starting byte number within each element to be used. The number of bytes in the field is specified by the f2 parameter, if present. If not specified, the number of bytes in the field is from f1 to the end of the element.

MAT MOVE (cont.)

In numeric-source-arrays, the entire value is always moved. In numeric-receiver-arrays, the entire value of the element is always replaced.

The counter-var

The counter-var may be used to specify the maximum number of elements to move.

NOTE: The move operation may be terminated by conditions other than the counter-var. If this occurs, the counter-var is set to the number of elements actually moved.

The pointer-array

The pointer-array can be used to reorder elements as they are moved. Ordered pointer-arrays are produced by MAT SORT and MAT MERGE. Typically, pointer-arrays are used in conjunction with one of these statements.

The pointer-array should contain a series of two-byte or four-byte subscripts used in accessing the source-array. If the source-array is a one dimensional array, the two-byte or four-byte subscript is treated as the binary representation of a single subscript (depending on the size of a pointer-array element). If the source-array is a two-dimensional array, a two-byte subscript is treated as containing two single byte binary subscripts, and a four-byte subscript is treated as containing two double-byte binary subscripts.

Pointer-arrays of this type are generated by MAT SORT and MAT MERGE statements.

When a pointer-array is specified, it is accessed to determine which element of the source-array to move next. That is, the first element moved is the element specified by the subscripts in the first element of the pointer-array. The next element to move is the element specified by the subscripts in the next element of the pointer-array, and so on.

A given element in the source-array may be referenced more than once by the pointer-array. If this occurs, the element is duplicated in the receiver-array.

The element location of the output of the move is not affected by the pointer-array.

If a pointer array element containing all HEX(00)'s is encountered, the move operation is terminated. The move operation is also terminated when the end of the pointer-array is reached.

MAT MOVE (cont.)

Pointer-array values are left unused if the move operation is terminated due to other conditions.

If a pointer array element containing all HEX(00) is encountered, the move operation is terminated.

NOTE: Some versions of NPL allow a two-dimensional array to be defined with more than 65535 rows or columns. However, subscripts greater than 65535 can not be stored in the pointer-array when a two-dimensional array is used.

Type Conversion

If the source-array and receiver-array are of different types, automatic conversion is performed between the NPL internal numeric format and an alpha format suitable for sorting. The alpha representation of numeric values requires an element length of 8 bytes to ensure that all possible values can be stored. Values are truncated or padded with spaces as required.

Byte one is used for the sign which may have the following decimal values:

9 - The mantissa and exponent are both positive.
8 - If a pointer array element containing all HEX(00) is encountered, the move operation is terminated. - the mantissa is positive but the exponent is negative.
1 - The mantissa and exponent are both negative.
0 - The mantissa is negative and the exponent is positive.

Byte two contains a representation of the exponent, high-order digit first. If the mantissa and exponent are the same sign, the exponent is stored in decimal form. Otherwise, the decimal complement form is used.

Remaining bytes are used to store the mantissa. The mantissa is stored in decimal format if the sign of the mantissa and exponent are the same. Otherwise, decimal complement format is used.

When converting from the alpha representation to a numeric value, an error occurs if the alpha data does not contain a valid representation of a numeric value.

MAT MOVE (cont.)

Examples:

```
0010 MAT MOVE A$(),L$(),N TO B$()
0010 MAT MOVE X$()(2,3) TO V$()
0010 MAT MOVE A(4,2),B$(1),100 TO C$()
0010 MAT MOVE G(),N$(5),H TO D()
0010 MAT MOVE L$(3,4)(5,8),Z$() TO S()
```

```
:0005 REM This example illustrates the field feature of MAT MOVE
:0010 DIM A$(10)10,C$(3,3)10
:0020 A$(1)="1234567890"          : REM Initialize source-array
0030 FOR X=2 TO 10
    :    A$(X)=A$(X-1)
    :    ROTATEC(A$(X),8)
    :    NEXT X
:0040 C=3                         : REM Initialize Counter
:0050 C$()=ALL("z")               : REM Initialize receiver-array
:0060 MAT MOVE A$()(2,3),C TO C$(2,2)(2,6): REM Move 3 byte field
                                           to a 6 byte field
:0070 LIST DIM * A$(,C$(

:RUN

DIM A$(10)10
(1)         "1234567890"        HEX(3132 3334 3536 3738 3930)
(2)         "2345678901"        HEX(3233 3435 3637 3839 3031)
(3)         "3456789012"        HEX(3334 3536 3738 3930 3132)
(4)         "4567890123"        HEX(3435 3637 3839 3031 3233)
(5)         "5678901234"        HEX(3536 3738 3930 3132 3334)
(6)         "6789012345"        HEX(3637 3839 3031 3233 3435)
(7)         "7890123456"        HEX(3738 3930 3132 3334 3536)
(8)         "8901234567"        HEX(3839 3031 3233 3435 3637)
(9)         "9012345678"        HEX(3930 3132 3334 3536 3738)
(10)        "0123456789"        HEX(3031 3233 3435 3637 3839)

DIM C$(3,3)10
(1,1)       "zzzzzzzzzz"        HEX(7A7A 7A7A 7A7A 7A7A 7A7A)
(1,2)       "zzzzzzzzzz"        HEX(7A7A 7A7A 7A7A 7A7A 7A7A)
(1,3)       "zzzzzzzzzz"        HEX(7A7A 7A7A 7A7A 7A7A 7A7A)
(2,1)       "zzzzzzzzzz"        HEX(7A7A 7A7A 7A7A 7A7A 7A7A)
(2,2)       "z234   zzz"        HEX(7A32 3334 2020 207A 7A7A)
(2,3)       "z345   zzz"        HEX(7A33 3435 2020 207A 7A7A)
(3,1)       "z456   zzz"        HEX(7A34 3536 2020 207A 7A7A)
(3,2)       "zzzzzzzzzz"        HEX(7A7A 7A7A 7A7A 7A7A 7A7A)
(3,3)       "zzzzzzzzzz"        HEX(7A7A 7A7A 7A7A 7A7A 7A7A)
```

Refer to MAT SORT and MAT MERGE statements for examples of the MAT MOVE statement as used in conjunction with these statements.

Compatibility Issues:

Use of 4-byte pointer arrays is supported in NPL Revision 4.0 or greater, and is not compatible with the Wang 2200.

MAT MOVE (cont.)

References:

MAT SORT
MAT MERGE

MAT* (Multiply)

General Form:

```
MAT array1 = array2 * array3
```

Where:

```
array1,array2,array3 = numeric-array names
```

Discussion:

The MAT* statement is used to multiply two arrays (array2,array3), resulting in a third array (array1).

The number of columns in array1 must equal the number of rows in array2. Array3 can not appear on both sides of the statement, otherwise an error is generated.

If array A has dimensions (L,M), and array B has dimensions (M,N) [note common dimension value], the statement MAT C=A*B is equivalent to:

```
0010 MAT REDIM C(L,N)
0020 FOR I=1 TO L
0030    FOR J=1 TO N
0040       S=0
0050       FOR K=1 TO M
0060          S=S+A(I,K)*B(K,J)
0070       NEXT K
0080       C(I,K)=S
0090    NEXT J
0100 NEXT I
```

Examples:

```
0010 MAT A = X * Y
0010 MAT C = A * B
```

Compatibility Issues:

References:

MAT PRINT

General Form:

```
MAT PRINT array-variable [{;} array-variable]...[{;}]
                          {,}                    {,}
```

Discussion:

The MAT PRINT statement is used to display the contents of the specified array-variable(s).

MAT PRINT displays the contents of the specified array-variable(s) in a row-by-row format, with each new row beginning on a new print line.

Specifying a trailing comma displays output in a column format. Specifying a trailing semicolon displays element rows continuously, with no blank spaces between elements in a row.

Examples:

```
0010 MAT PRINT A$
0010 MAT PRINT A;
0010 MAT PRINT A,A$,B
0010 MAT PRINT X$

:0010 DIM A$(3,3)16,B(3,3)
:0020 MAT INPUT A$,B
:0030 MAT PRINT A$,B
:RUN
? THIS,IS,AN,EXAMPLE,OF,THE,MAT,INPUT,STATEMENT
? 1,2,3,4,5,6,7,8,9

THIS            IS              AN
EXAMPLE         OF              THE
MAT             INPUT           STATEMENT

 1               2               3
 4               5               6
 7               8               9
```

Compatibility Issues:

References:

MAT READ

General Form:

```
    MAT READ array-variable [(dim1[,dim2]) [length]]
                            [,array-variable [(dim1[,dim2]) [length]]]...
```

Where:

```
    dim1,dim2 = numeric-expressions specifying new dimensions of the
                array.

    length    = expression specifying the length of each element in
                an alpha-array.  Default length is 16.
```

Discussion:

The MAT READ statement is used to assign a list of values from a DATA statement to the arrays specified.

Data is assigned row-by-row to each array, continuing left to right through the arrays. The process uses the data values sequentially, beginning with the DATA statement with the lowest line-number, using each value, then those from the next DATA statement, and so on until the elements of all arrays have been filled or the data is exhausted. An error occurs if the data is exhausted first. An error also occurs if the value being transferred does not match the variable type required by the list of arrays.

Examples:

```
0010 MAT READ A$,B$
0010 MAT READ A$
0010 MAT READ B,C
```

Compatibility Issues:

References:

DATA

MAT REDIM

General Form:

```
MAT REDIM array-variable (dim1[,dim2]) [length]
                  ,array-variable (dim1[,dim2]) [length]]...
```

Where:

```
dim1,dim2 = numeric-expressions specifying new dimensions of the
            array.

length    = expression specifying the length of each element in
            an alpha-array.  Default length is 16.
```

Discussion:

The MAT REDIM statement is used to redimension the specified array according to the specified dimension parameters. In addition, if the redimensioned array is an alpha-array, the length of the element may optionally be specified.

MAT REDIM may be used to expand statically allocated arrays beyond their initially allocated size, provided:

1. There is sufficient memory to allocate both the old variable and the new large allocation.

2. There may not be any pending /POINTER or stack references to the variable.

If either condition is not met, an error 304 (cannot expand any variable) occurs. (This is a recoverable error.)

The following example illustrates condition (1) above:

MAT REDIM (cont.)

```
0010 ; SYNALLOC - Example for Dynamically Allocating Arrays
0020 DIM X,Y
0030 DIM A$(0)0
0040 FUNCTION 'GetSize(/POINTER Elements for Element";
0050 PRINT "Enter Number of elements for Array";
0060 INPUT Elements
0070 PRINT "Enter Number of Elements for Array";
0080 INPUT Size
0090 RETURN (0)
0100 END FUNCTION
0110 IF 'GetSize(X,Y)=0
   :   MAT REDIM A$(X)Y
   : END IF
0120 LIST DIM A$(
:RUN
Enter Number of Elements for Array ? 1024
Enter Length of each Array Element ? 32
DIM A$(1024)32
```

Condition 2 avoids errors caused by references to the variable at the old address. Because the MAT REDIM statement expands the array, and it moves it to a new address, the assignment would reference the old copy of the array unless the /POINTER addresses are corrected retroactively.

Newly allocated space in the variable contains either blanks, if a string, or zeroes, for numerics. Previously allocated space is unchanged.

This extended capability of MAT REDIM also applies to implied or explicit array redimensioning operations in the matrix math statements:

MAT =	MAT +	MAT -	MAT *
MAT SCALAR	MAT IDN	MAT CON	MAT ZER

The scope of statically allocated arrays may be PUBLIC, module, private or local to a function.

NOTE: Recursive and scalar variables may not be specified in MAT REDIM statements.

MAT REDIM (cont.)

Examples:

```
0010 MAT REDIM Z(5,5)
0010 MAT REDIM X(9,Y),N(10,10)
0010 MAT REDIM K$(6,12)10
0010 MAT REDIM V$(4,4)12,X$(8,4)10,Z$(12)8

10 DIM A$(0)20, C(0)
   : READ X                      :; get list size
   : MAT REDIM A$(X)20,          ;expand table of names
          C(X)                   :; and amounts.
```

Compatibility Issues:

Revisions prior to NPL Release IV do not allow MAT REDIM to redimension an array larger than that of the original array.

References:

MAT SEARCH

General Form:

```
    MAT SEARCH [ELEMENT] {alpha-variable1[<[s][,[n]]>],} rel-op
                         {literal-string1,            }

          alpha-value TO receiver-variable [STEP numeric-expression]
```

Where:

```
    s                 = numeric-expression

    n                 = numeric-expression

    rel-op            = {<, <=, =, >, >=, <>}

    alpha-value       = {alpha-variable     }
                        {literal            }

    receiver-variable = {alpha-variable       }
                        {numeric-array        }
                        {numeric-variable     }
                        {numeric-array element}
```

Discussion:

The MAT SEARCH statement searches alpha-variable1 for substrings which satisfy the specified relational operator when compared to alpha-value, and places the addresses of these substrings in the receiver-variable.

If the receiver-variable is an alpha-variable, each address is stored as a two-byte binary value that indicates the starting position in alpha-variable1. If the receiver-variable is a numeric-array, each address is stored as a numeric value with the first address placed in element one of the array, the second placed in element two, and so on.

If the receiver-variable is a numeric-variable or a numeric-array element, only the first location or element in the string which matches the search condition is placed in the variable, or a zero value is placed in the variable if no match is found.

MAT SEARCH (cont.)

The keyword ELEMENT determines the actual values returned for "address". If ELEMENT is not specified, the address is the first byte number in alpha-variable1 that corresponds to the start of a matching substring. The ELEMENT keyword is normally used when searching an array of fixed size elements for a value that can occur at only one position in an element. In this case, the STEP value would be the size of the array element and the value returned in the receiver-variable is an element number rather than a byte address. The non-zero values returned by MAT SEARCH and MAT SEARCH ELEMENT are related by the formula:

$$e = (b-1)/s+1$$

where:

e is the element number returned by MAT SEARCH ELEMENT

b is the byte address returned by MAT SEARCH

s is the STEP value

If alpha-variable1 is an array, element boundaries are ignored and the address is specified starting from byte one of the array.

The value of zero is placed in the receiver-variable just after the address of the last valid substring found.

When MAT SEARCH is executed, substrings of alpha-variable1 are tested against the alpha-value for the relation specified. The length of the substring tested is the length of alpha-value.

NOTE: Trailing spaces in alpha-value are ignored unless alpha-value is a literal or STR() function.

The portion of alpha-variable1 SEARCHed can be modified by use of the "s" and "n" parameters. The "s" parameter is a numeric-expression which specifies the position in alpha-variable1 to begin the SEARCH operation. The "n" parameter is a numeric-expression which specifies the number of bytes of alpha-variable1 (starting at byte 1 if "s" parameter is not specified) to be searched.

MAT SEARCH (cont.)

The optional "STEP" parameter is used to specify the number of bytes to skip in alpha-variable1 in determining the starting address of the next field to search. The "STEP" parameter must be a positive integer. If no "STEP" parameter is indicated, a default of "1" is used. The STEP parameter is useful for searching alpha-variables which contain field-oriented data. For example, if an alpha-variable contains a series of 8-byte names, and the purpose of the search was to find a particular name, a STEP value of 8 would be appropriate. Failure to specify an appropriate STEP value could result in erroneous data being selected by the search operation and in more lengthy execution time.

For example, assume that an array contains elements of length 20 where each element contains a person’s name (first name first). The purpose of this program is to extract and print all names where the first name is some variation of the name "MARY" (MARIE, MARIANNE, etc.).

```
:05 DIM A$(5)20
:10 DIM X$(5)2                                :REM POINTER VARIABLE
:15 A$(1)="MARY SMITH"
  : A$(2)="SUSAN JONES"
  : A$(3)="MARIANNE JACKSON"
  : A$(4)="SALLY MARGOLAN"
  : A$(5)="JANE ADAMS"
:20 MAT SEARCH A$(),="MAR" TO X$() STEP 20    :REM SEARCH FOR NAMES
                                               STARTING WITH "MAR"

:30 FOR X=1 TO 5
    :IF X$(X)=HEX(0000) THEN 90               :REM NO MORE POINTERS
    :   PRINT STR(A$(),VAL(X$(X),2),20)       :REM PRINT THE NAME
                                               BASED ON THE POINTER
                                               VALUE
    :   NEXT X                                :REM NEXT POINTER VALUE

:80 GOTO 100
:90 X=5: NEXT X                               :REM CLEAR THE LOOP
:100 REM DONE

:RUN

MARY SMITH
MARIANNE JACKSON
```

In this case, the pointer variable contains two addresses in binary - (0001) & (0029) which are converted to decimal byte addresses by the VAL function in line 30.

MAT SEARCH (cont.)

NOTE: If a step value was not specified in line 20, the "MAR" in the name SALLY MARGOLAN would also meet the search criterion and the program would print:

MARY SMITH
MARIANNE JACKSON
MARGOLAN JANE A

In this case, the pointer variable contains three addresses in binary - (0001), (0029) and (0043) which are converted to decimal byte addresses by the VAL function in line 30.

The above example could be modified to use the ELEMENT keyword and a numeric-array as the receiver-variable as follows:

```
:10 DIM X(5)                      :REM POINTER VARIABLE
:15 A$(1)="MARY SMITH"
  : A$(2)="SUSAN JONES"
  : A$(3)="MARIANNE JACKSON"
  : A$(4)="SALLY MARGOLAN"
  : A$(5)="JANE ADAMS"
:20 MAT SEARCH ELEMENT A$(),="MAR" TO X() STEP 20
                                  :REM SEARCH FOR NAMES STARTING WITH "MAR"
:30 FOR X=1 TO 5
  :IF X(X)0 THEN DO
  :        PRINT A$(X(X))         :REM PRINT THE NAME
  :        NEXT X                 :REM NEXT POINTER VALUE
  :                 ENDDO
  :   ELSE
  :        NEXT CLEAR             :REM NO MORE POINTERS
:100 REM DONE

:RUN

    MARY SMITH
MARIANNE JACKSON
```

NOTE: Use of the ELEMENT keyword in conjunction with a numeric-array as a receiver-variable significantly simplifies the code. In particular, note that elements of A$ array which match can be referenced using the receiver-variable (X()) as a subscript.

MAT SEARCH (cont.)

Examples:

```
0010 MAT SEARCH A$(),=B$ TO L$
0010 MAT SEARCH STR(D$(),1,100), = STR(F$,1,2) TO B$()
0010 MAT SEARCH D$(),=M$() TO STR(P$(),30,10)
0010 MAT SEARCH A$(),<B$() TO C$()
0010 MAT SEARCH X$(),<=Y$ TO Z$ STEP 20
0010 MAT SEARCH A$(),=B$ TO L$
0010 MAT SEARCH STR(D$(),1,100), = STR(F$,1,2) TO B()
0010 MAT SEARCH D$(),=M$() TO STR(P$(),30,10)
0010 MAT SEARCH A$(),<B$() TO C$()
0010 MAT SEARCH ELEMENT X$(),<=Y$ TO Z() STEP 20
```

Compatibility Issues:

Use of numeric-arrays to receive the results of MAT SEARCH is supported only in NPL Revision 3.0 or greater.

Use of numeric-arrays to receive the results of MAT SEARCH is not supported on the Wang 2200.

Use of the ELEMENT keyword is supported only in NPL Revision 3.0 or greater and is not supported on the Wang 2200.

References:

MAT SORT

General Form:

```
     MAT SORT source-array[(f1[,f2])] TO temp-variable, pointer-array
```

Where:

```
     source-array  = alpha-array

     (f1,f2)       = optional byte range (start & length) which de-
                     fines a field within each alpha-array element:

                     f1 = numeric expression which specifies the
                          starting position of the field.

                     f2 = numeric expression which specifies the
                          length of the field.

     temp-variable = an alpha-variable used as a work area which must
                     be dimensioned with two bytes for every element
                     in the source-array, or four bytes per element if
                     the pointer-array element length is four bytes.

     pointer-array = alpha-array of two or four byte elements.
```

Discussion:

The MAT SORT statement is used to sort the specified source-array in ascending order. Sorting is performed on a binary basis for each element of the source-array. Sorting may be performed on a specified field within elements of the source-array by specifying the starting byte number and number of bytes to use as the f1 and f2 parameters. If only f1 is specified, the field is comprised of all bytes from the f1 position to the end of the element.

MAT SORT (cont.)

Output of the MAT SORT operation consists of a pointer-array where each element of the pointer-array contains a subscript for one element of the source-array in order of the specified sort.

NOTE: NPL allows a two-dimensional array to be defined with more than 255 rows or columns (though the total elements may not exceed 65535). However, subscripts greater than 255 cannot be stored in the pointer-array when a two-dimensional array is used. If such an array is used, it must be redimensioned (REDIM) before sorting.

The pointer-array must be dimensioned with at least as many elements as the source-array with a length of two or four bytes. A four-byte element length is required to sort one-dimensional arrays with more than 65535 elements, or two-dimensional arrays with more than 255 rows or columns, but may also be used for smaller source-arrays. Two-dimensional arrays with more than 65535 rows or columns cannot be sorted by MAT SORT.

For one-dimensional source-arrays, the subscript is a two or four byte binary number, depending on the length of the pointer-array elements. For two dimensional source-arrays, each subscript is stored as either a one-byte binary number within the two-byte pointer-array element, or as a two-byte binary number within the four-byte pointer-array element. If the number of pointer-array elements exceeds the number of source-array elements, the first unused pointer-array element is set to a value of all HEX(00).

NOTE: Some versions of NPL allow a two-dimensional array to be defined with more than 65535 rows or columns. However, subscripts greater than 65535 can not be stored in the pointer-array when a two-dimensional array is used.

The pointer-array produced by MAT SORT can be used in several ways:

1. It can be used in conjunction with a MAT MOVE statement to MOVE the source-array to an output array in order of the pointer-array.

MAT SORT (cont.)

2. It can be used to directly access the source-array by use of the subscripts stored. This permits the programmer to access the source-array in more complex ways. One such use would be to access the source-array in descending order.

For example, refer to the following:

```
:10 DIM I$(50)8, T$100, P$(50)2: REM T$=TEMP; P$(=POINTER
:20 MAT INPUT I$
:30 MAT SORT I$() TO T$,P$()
:40 FOR X=50 TO 1 STEP -1
:    PRINT I$(VAL(P$(X),2))
:    NEXT X
```

prints each element of I$() in descending sequence.

NOTE: This example uses a one-dimensional source array. For a two-dimensional array, the logic for accessing the source-array directly based on the subscripts stored in the pointer-array is slightly different because each element of the pointer-array contains subscripts for both row and column rather than a single two-byte subscript:

```
:10 DIM I$(25,2)8, T$100, P$(50)2: REM T$=TEMP; P$(=POINTER
:20 MAT INPUT I$
:30 MAT SORT I$() TO T$,P$()
:40 FOR X=50 TO 1 STEP -1
:    PRINT I$(VAL(STR(P$(X),,1)),VAL(STR(P$(X),2,1)))
:    NEXT X
```

MAT SORT is also frequently used in conjunction with MAT MERGE to produce a sorted output array based on multiple sorted source arrays.

Only alpha-arrays can be specified as the source-array to a MAT SORT operation. If sorting of numeric arrays is required, the numeric array must first be moved to an alpha-array with the same dimensions by use of the MAT MOVE statement. Then, after the sort, the pointer-array indexes the numeric-array in sorted order. Example 2 below illustrates this process. Refer to MAT MOVE for further details on conversion between numeric and alpha arrays.

MAT SORT (cont.)

Examples:

```
10 MAT SORT A$() TO T$,P$()
10 MAT SORT A$() (2,3) TO T$,P$()

:0010 DIM A$(5,3)16,T$30,P$(15)2,X$(5,3)16
:0020 MAT INPUT A$
:0030 MAT SORT A$() TO T$,P$()
:0040 LIST DIM *A$(,P$(           :REM List contents of array variables in the
                                   range of A$ to P$.
:0100 MAT MOVE A$(),P$(1) TO X$(1,1)
:0120 LIST DIM *X$(     :REM List contents of resulting move-array
:0200 END
:RUN
```

Results: (after MAT INPUT A$ operation is performed)

```
DIM A$(5,3)16
(1,1)       "AAAAAAAA        "  HEX(4141 4141 4141 4141 2020 2020 2020 2020)
(1,2)       "CCCCCCCC        "  HEX(4343 4343 4343 4343 2020 2020 2020 2020)
(1,3)       "JJJJJJJJJJ      "  HEX(4A4A 4A4A 4A4A 4A4A 4A4A 2020 2020 2020)
(2,1)       "EEEEEEEEEEE     "  HEX(4545 4545 4545 4545 4545 4520 2020 2020)
(2,2)       "IIIIIIII        "  HEX(4949 4949 4949 4949 2020 2020 2020 2020)
(2,3)       "QQQQQQQQQQQQQ   "  HEX(5151 5151 5151 5151 5151 5151 5120 2020)
(3,1)       "UUUUUUU         "  HEX(5555 5555 5555 5520 2020 2020 2020 2020)
(3,2)       "IIIIIIIIII      "  HEX(4949 4949 4949 4949 4949 2020 2020 2020)
(3,3)       "MMMMMMMMMM      "  HEX(4D4D 4D4D 4D4D 4D4D 4D4D 2020 2020 2020)
(4,1)       "KKKKKKKKKK      "  HEX(4B4B 4B4B 4B4B 4B4B 4B4B 2020 2020 2020)
(4,2)       "PPPPPPPPPPPPP   "  HEX(5050 5050 5050 5050 5050 5050 5020 2020)
(4,3)       "XXXXXXXXXXX     "  HEX(5858 5858 5858 5858 5858 5820 2020 2020)
(5,1)       "XXXXXXXXXXXXX   "  HEX(5858 5858 5858 5858 5858 5858 5820 2020)
(5,2)       "JJJJJJJJJJ      "  HEX(4A4A 4A4A 4A4A 4A4A 4A4A 2020 2020 2020)
(5,3)       "111111111111    "  HEX(3131 3131 3131 3131 3131 3131 2020 2020)

DIM P$(15)2
(1)         ".."                HEX(0503)
(2)         ".."                HEX(0101)
(3)         ".."                HEX(0102)
(4)         ".."                HEX(0201)
(5)         ".."                HEX(0202)
(6)         ".."                HEX(0302)
(7)         ".."                HEX(0103)
(8)         ".."                HEX(0502)
(9)         ".."                HEX(0401)
(10)        ".."                HEX(0303)
(11)        ".."                HEX(0402)
(12)        ".."                HEX(0203)
(13)        ".."                HEX(0301)
(14)        ".."                HEX(0403)
(15)        ".."                HEX(0501)
```

MAT SORT (cont.)

NOTE: A$() is the source-array
P$() is the pointer-array
T$ is the temp-variable

```
DIM X$(5,3)16
(1,1)       "111111111111    "  HEX(3131 3131 3131 3131 3131 3131 2020 2020)
(1,2)       "AAAAAAAA        "  HEX(4141 4141 4141 4141 2020 2020 2020 2020)
(1,3)       "CCCCCCCC        "  HEX(4343 4343 4343 4343 2020 2020 2020 2020)
(2,1)       "EEEEEEEEEEE     "  HEX(4545 4545 4545 4545 4545 4520 2020 2020)
(2,2)       "IIIIIIII        "  HEX(4949 4949 4949 4949 2020 2020 2020 2020)
(2,3)       "IIIIIIIIII      "  HEX(4949 4949 4949 4949 4949 2020 2020 2020)
(3,1)       "JJJJJJJJJJ      "  HEX(4A4A 4A4A 4A4A 4A4A 4A4A 2020 2020 2020)
(3,2)       "JJJJJJJJJJ      "  HEX(4A4A 4A4A 4A4A 4A4A 4A4A 2020 2020 2020)
(3,3)       "KKKKKKKKKK      "  HEX(4B4B 4B4B 4B4B 4B4B 4B4B 2020 2020 2020)
(4,1)       "MMMMMMMMMM      "  HEX(4D4D 4D4D 4D4D 4D4D 4D4D 2020 2020 2020)
(4,2)       "PPPPPPPPPPPPP   "  HEX(5050 5050 5050 5050 5050 5050 5020 2020)
(4,3)       "QQQQQQQQQQQQQ   "  HEX(5151 5151 5151 5151 5151 5151 5120 2020)
(5,1)       "UUUUUUU         "  HEX(5555 5555 5555 5520 2020 2020 2020 2020)
(5,2)       "XXXXXXXXXXX     "  HEX(5858 5858 5858 5858 5858 5820 2020 2020)
(5,3)       "XXXXXXXXXXXXX   "  HEX(5858 5858 5858 5858 5858 5858 5820 2020)
```

NOTE: X$() is the move-array which contains the A$() array in sorted order after being filled using the MAT MOVE statement.

Example 2:

```
:0005 REM This example allows input into numeric array A, convert
      array A to alpha array A$, sort array A$, and move the sorted
      results into numeric array B
:0010 DIM A(5,4),B(5,4),A$(5,4)8,T$40,P$(20)2
:0020 MAT INPUT A
:0025 MAT MOVE A() TO A$()      :REM Convert to alpha
:0030 MAT SORT A$() TO T$,P$()
:0040 MAT MOVE A(),P$() TO B()  :REM Convert to numeric and
                                 move in sorted order
:0050 MAT PRINT A
:0060 MAT PRINT B
:0070 LIST DIM * P$(            :REM Display the resulting pointer array

:RUN
```

NOTE: A() is source array entered in response to MAT INPUT:

```
 10.5          -9.4            0              12
298.784       -48.89          .001           -1.9
-10.5          98.45          -.001           76
 28            -9.98           .87            87
 16.5           7.8           -.78            9
```

MAT SORT (cont.)

NOTE: B() is sorted output:

```
-48.89         -10.5          -9.98          -9.4
-1.9           -.78           -.001           0
 .001           .87            7.8            9
 10.5           12             16.5           28
 76             87             98.45          298.784

DIM P$(20)2
(1)        ".."               HEX(0202)
(2)        ".."               HEX(0301)
(3)        ".."               HEX(0402)
(4)        ".."               HEX(0102)
(5)        ".."               HEX(0204)
(6)        ".."               HEX(0503)
(7)        ".."               HEX(0303)
(8)        ".."               HEX(0103)
(9)        ".."               HEX(0203)
(10)       ".."               HEX(0403)
(11)       ".."               HEX(0502)
(12)       ".."               HEX(0504)
(13)       ".."               HEX(0101)
(14)       ".."               HEX(0104)
(15)       ".."               HEX(0501)
(16)       ".."               HEX(0401)
(17)       ".."               HEX(0304)
(18)       ".."               HEX(0404)
(19)       ".."               HEX(0302)
(20)       ".."               HEX(0201)
```

Compatibility Issues:

On the Wang 2200 MVP, if the array to be sorted contains duplicate values, the order in which duplicate elements appear in the pointer-array is not defined. The NPL version of the sort guarantees that the order of identically-valued elements is preserved in the pointer-array.

It is very unlikely that a program might depend on the exact order produced by the Wang 2200 MVP when keys are identical, or might try to use the contents of the work-variable (which are also undefined after the SORT operation) for some obscure purpose. Programs which do so will fail.

Use of 4-byte pointer arrays is supported in NPL Revision 4.0 or greater, but is not compatible with the Wang 2200.

MAT SORT (cont.)

References:

MAT MOVE
MAT PRINT

MAT TRN

General Form:

```
MAT numeric-array1 = TRN(numeric-array2)
```

Discussion:

The MAT TRN statement used to replace numeric-array1 with the transpose of numeric-array2.

The receiving numeric-array1 is redimensioned to the same dimensions as the transpose of numeric-array2. Numeric-array1 cannot appear on both sides of the equation or an error is generated.

Transposition takes place element by element. For example, element (2,3) of numeric-array2 becomes element (3,2) of numeric-array1.

If array A has dimension (L,M), the statement MAT B=TRN(A) is equivalent to:

```
0010 MAT REDIM B(M,L)
   : FOR I=1 TO L
   :    FOR J=1 TO M
   :       B(J,I)=A(I,J)
   :    NEXT J
   : NEXT I
```

Examples:

```
0010 MAT A = TRN(B)
0010 MAT C = TRN(X)
0010 MAT B1 = TRN(Y)
```

Compatibility Issues:

References:

MAT ZER

General Form:

```
     MAT numeric-array = ZER [(dim1[,dim2])]
```

Where:

```
     dim1, dim2 = numeric-expressions specifying new dimensions of
                  the numeric-array.
```

Discussion:

The MAT ZER statement is used to set all elements of the specified numeric-array to the numeric value of 0. In addition, the specified numeric-array is optionally redimensioned according to the new dimension parameters.

If the redimensioning parameters are specified, the total elements must be less than or equal to the original number of elements.

Examples:

```
0010 MAT Z = ZER
0010 MAT X = ZER(5,10)
0010 MAT P = ZER(Q,S)
0010 MAT K = ZER(15)
```

Compatibility Issues:

References:

MAT Addition

General Form:

```
MAT numeric-array1 = numeric-array2 + numeric-array3
```

Discussion:

The MAT addition statement is used to add together all elements of two equally dimensioned numeric-arrays. The numeric sum is stored in the corresponding elements of numeric-array1. Numeric-array1 is automatically redimensioned to the dimensions of the arrays that are added to it. Either numeric-arrays 2 or 3 may appear on both sides of the equation.

Examples:

```
0010 MAT Z = X + Y
0010 MAT X = X + A
0010 MAT P = P + P

:0010 DIM X(2,3),Y(3,2),Z(3,2)
:0020 MAT INPUT X
:0030 PRINT "AFTER INPUT MAT X = "
:0040 MAT PRINT X
:0050 MAT Y=X
:0060 MAT Z=X+Y
:0070 PRINT "AFTER MAT +, MAT Z = "
:0080 MAT PRINT Z
:RUN

AFTER INPUT MAT X =

 1              2              3
 4              5              6

AFTER MAT +, MAT Z =

 2              4              6
 8              10             12
```

Compatibility Issues:

References:

MAT Assignment

General Form:

```
MAT numeric-array1 = numeric-array2
```

Discussion:

The MAT assignment statement is used to set all elements of numeric-array1 equal to the values of the corresponding elements in numeric-array2. Numeric-array1 is automatically redimensioned to the dimensions of numeric-array2.

Examples:

```
0010 MAT Z = X
0010 MAT X = A
0010 MAT P = Y

:0010 DIM X(2,3),Y(3,2),Z(3,2)
:0020 MAT INPUT X
:0030 PRINT "AFTER INPUT MAT X = "
:0040 MAT PRINT X
:0050 MAT Y=X
:0060 PRINT "AFTER MAT =, MAT Y = "
:0070 MAT PRINT Y
:RUN

AFTER INPUT MAT X =

 1              2              3
 4              5              6

AFTER MAT =, MAT Y =

 1              2              3
 4              5              6
```

Compatibility Issues:

References:

MAT Scalar Multiplication

General Form:

```
MAT numeric-array1 = (numeric-expression) * numeric-array2
```

Discussion:

MAT scalar multiplication is used to multiply all elements of numeric-array2 by the value of a numeric-expression. The numeric-array of multiplicative products is stored in the elements of numeric-array1. Numeric-array1 is automatically redimensioned to the dimensions of numeric-array2.

Examples:

```
0010 MAT Z = (2) * X
0010 MAT X = (D+P*2) * Y
0010 MAT P = (X-4) * N

:0010 DIM X(2,3),Y(3,2),Z(3,2)
:0020 MAT INPUT X
:0030 PRINT "AFTER INPUT MAT X = "
:0040 MAT PRINT X
:0050 A=4
:0060 MAT Z=(A)*X
:0070 PRINT "AFTER MAT *, MAT Z = "
:0080 MAT PRINT Z
:RUN

AFTER INPUT MAT X =
 1              2              3
 4              5              6

AFTER MAT *, MAT Z =
 4              8              12
 16             20             24
```

Compatibility Issues:

References:

MAT Subtraction

General Form:

```
MAT numeric-array1 = numeric-array2 - numeric-array3
```

Discussion:

The MAT subtraction statement is used to subtract all elements of numeric-array3 from the elements of numeric-array2, given that numeric-array2 and numeric-array3 are equally dimensioned. The difference is stored in the corresponding elements of numeric-array1. Numeric-array1 is automatically redimensioned to the dimensions of the arrays that are involved in the subtraction.

Examples:

```
0010 MAT Z = X - Y
0010 MAT X = X - A
0010 MAT P = Q - T

:0010 DIM X(2,3),Y(3,2),Z(3,2)
:0020 MAT INPUT X
:0030 PRINT "AFTER INPUT MAT X = "
:0040 MAT INPUT Y
:0050 PRINT "AFTER INPUT MAT Y = "
:0060 MAT PRINT Y
:0070 MAT Z=Y-X
:0080 PRINT "AFTER MAT -, MAT Z = "
:0090 MAT PRINT Z
:RUN

AFTER INPUT MAT X =

 1               2               3
 4               5               6

AFTER INPUT MAT Y =

 3               5               7
 9               11              13

AFTER MAT -, MAT Z =

 2               3               4
 5               6               7
```

Compatibility Issues:

References:

MAX Function

General Form:

```
MAX ({numeric-expression} [,{numeric-expression}]...)
     {numeric-array-name}   {numeric-array-name}
```

Discussion:

The MAX function returns the highest numeric value from the argument list. The argument list can consist of any number of numeric-expressions. If a numeric-array is specified, each element of the array is considered a separate argument for the function. This is valid wherever a numeric-expression is legal.

Examples:

```
0010 X = MAX(A,Z)
0010 W = MAX(0,X(),Q)
0010 IF A = MAX(X(),10) THEN A = A + 1
0010 FOR I = 1 TO MAX(Q,20)
0010 X=MAX(Z/A,B(1)+3)
0010 High_Month=MAX(Monthly_Sales())
:0010 PRINT MAX(-100,2)
: PRINT MAX(59.2,2)
: PRINT MAX(50,-51,49,(29-10))
:RUN
 2
 59.2
 50
```

Compatibility Issues:

References:

MIN Function

MIN Function

General Form:

```
MIN ({numeric-expression} [,{numeric-expression}]...)
     {numeric-array-name}    {numeric-array-name}
```

Discussion:

The MIN function returns the lowest numeric value from the argument list. The argument list can consist of any number of numeric-expressions. If a numeric-array is specified, each element of the array is considered a separate argument for the function. This is valid wherever a numeric-expression is legal.

Examples:

```
0010 X = MIN(A,Z)
0010 W = MIN(0,X(),Q)
0010 IF A = MIN(X(),10) THEN A = A + 1
0010 FOR I = 1 TO MIN(Q,20)
0010 Y=MIN(J*K,VAL(A$),L(3)+G)
0010 Low_Month=MIN(Monthly_Sales())
:0010 PRINT MIN(-100,2)
    : PRINT MIN(-2,2)
    : PRINT MIN(50,141,49,(29-10))
:RUN
-100
-2
 19
```

Compatibility Issues:

References:

MAX Function

MOD Function

General Form:

```
MOD (numeric-expression1,numeric-expression2)
```

Discussion:

The MOD function computes the remainder (modulus) when numeric-expression1 is divided by numeric-expression2. This is valid wherever a numeric-expression is legal.

NOTE: If X> 0, then 0 < = MOD(Y,X) < X
If X< 0, then X < MOD(Y,X) < = 0
If X= 0, then MOD(Y,X) = Y
In all cases, MOD(Y,X)= Y-INT(Y/X)*X.

Examples:

```
0010 Q = MOD(Y,10)
0010 R = MOD(15.6,-8)
0010 E = MOD(-3,X)

:0010 Q = MOD(8,3)
:0020 PRINT Q

:RUN

 2
```

Compatibility Issues:

References:

MODULE Command

General Form:

```
MODULE   {alpha-variable}
         {literal-string}
```

Discussion:

The MODULE command may be used in Immediate Mode to change the current LIST MODULE value to another module. This allows the user to view (using LIST commands) and modify (using LOAD or editing commands) program text in a module other than the one currently executing.

NOTE: The statement stays in effect only until the program is continued or restarted.

Examples:

```
:MODULE "AccountsReceivableSystemMenu": LIST
:MODULE "FIELDPCK"
:MODULE " " :; back to main module
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

LOAD
Refer to Chapter 6, "Debugging Code"

MOVE

General Form:

Form 1:

```
MOVE T [file-number,    ] TO T [file-number,   ][LS=val1,][END=val2]
       [disk-address, ]     [disk-address, ]
       [<address-var>,]     [<address-var>,]
```

Form 2:

```
MOVE T [file-number,  ] filename1 TO T
       [disk-address, ]
       [<address-var>,]
                    [file-number,   ]  [([filename2])]
                    [disk-address,  ]  [space        ]
                    [<address-var>, ]
```

Where:

```
val1      = a numeric-expression which represents the required
            size of the Catalog Index whose value is from 1 to
            255.

val2      = a numeric-expression which represents the highest sec-
            tor address in the Catalog area. The expression must
            be less than or equal to the highest sector address
            available on disk.

space     = a numeric-expression indicating the number of extra
            sectors to be saved with the file on the destination
            diskimage.

filename1 = literal or alpha-variable containing the name of the
            file to move.

filename2 = literal or alpha-variable containing the name of the
            file to be overwritten.
```

MOVE (cont.)

Discussion:

The MOVE statement is used to copy an entire diskimage (Form 1) or an individual cataloged disk file (Form 2) to a second specified diskimage.

Form 1:

In Form 1, MOVE copies the contents of one diskimage to another diskimage but excludes scratched files and temporary files from the copy. The catalog index is rebuilt on the destination diskimage, and the input diskimage is not modified. The destination diskimage is scratched before any files are moved. If the optional LS and/or END parameters are specified, these values are applied to the output diskimage. Otherwise, the values from the input diskimage are used. If LS and/or END are specified, the output diskimage is created with the alternate hashing algorithm (equivalent to SCRATCHDISK’). Otherwise, the hashing algorithm of the input diskimage is used.

Form 1 of the MOVE statement also updates file trailer fields for file name, status, and type.

Form 2:

In Form 2, the MOVE statement copies a single cataloged file from one diskimage to another.

MOVE may be used to add a new file to the output diskimage by specification of the space parameter. Space determines the number of extra sectors to add to the new file. If space is zero or no parenthesis expression is given, the new file is created as the size of the input file. Alternatively, the input file may be written over an existing scratched file in the output diskimage. The filename to overwrite is specified by the filename2 parameter. If filename2 is not present (a set of empty parentheses), the filename of the input file is used.

The Date/Time stamp of the file is not affected by the MOVE statement. The Date/Time stamp of the file is the same after the MOVE operation as before the MOVE operation.

MOVE (cont.)

Examples:

```
0010 MOVE T TO T/D20,
0010 MOVE T/D12, TO T/D23, LS=123,
0010 MOVE T#1, TO T#2, END=34600
0010 MOVE T<A$>, TO T/D12, LS=47, END=98400

0010 MOVE T/D22,"ARCU3" TO T<B$>,()
```

This statement copies file ARCU3 from diskimage D22 to the diskimage whose address is contained in variable B$ overwriting scratched file ARCU3.

```
0010 MOVE T#1,"SACM1999" TO T/D20,("SACM1200")
```

This statement copies file SACM1999 from the diskimage selected as file-number 1 to diskimage D20 overwriting file SACM1200. The name of the file on D20 is SACM1999.

```
0010 MOVE T/D31,B$ TO T
```

This statement copies the file whose name is contained in variable B$ from diskimage D31 to the diskimage selected as file-number 0, creating it as a new file with the same size and the same name.

```
0010 MOVE T/D23,"SASM3" TO T#2,(4)
```

This statement copies file SASM3 from diskimage D23 to the diskimage selected as file-number 2, creating SASM3 as a new file with 4 additional sectors.

Compatibility Issues:

The optional LS and END parameters for form 1 of MOVE are supported only in NPL Revision 3.0 or greater.

Update of the file trailer sector for form 1 of MOVE is supported only in NPL Revision 3.0 or greater and is not supported on the Wang 2200.

Use of the address-var parameter is supported only on NPL Revision 3.0 or greater and is not supported on the Wang 2200.

References:

SCRATCH DISK

MOVE END

General Form:

```
MOVE END T [file#,        ] = numeric-expression
           [disk-address, ]
           [<address-var>,]
```

Discussion:

The MOVE END statement is used to alter the size of the catalog area of a diskimage. The diskimage is specified along with a numeric-expression indicating the physical number of sectors.

MOVE END may be used to increase or decrease the catalog area of a diskimage. However, the catalog area may never be decreased to less than the Current End. On some operating systems, MOVE END may physically increase or decrease the size of the diskimage file.

Examples:

```
0010 MOVE END T=19583
0010 MOVE END T/D20,=52607
0010 MOVE END T<A$>,=52607
0010 MOVE END T#3,=X2*512
```

Compatibility Issues:

The MOVE END command has been extended in NPL. In addition to the Wang Basic-2 function of defining the amount of a diskimage to be usable by the cataloged commands, in NPL the MOVE END statement also performs the function of EXTENDING or REDUCING the physical size of a diskimage file. The diskimage file is extended or truncated to the size of the specified END= expression, provided physical disk space is available.

NOTE: This feature does not apply to "raw" diskettes.

On some operating systems, when used to decrease the size of a diskimage file, MOVE END does not release space. Refer to the appropriate NPL Supplement for hardware and operating system-specific details.

MOVE END (cont.)

Some operating systems may place limitations on the amount an existing diskimage file can be expanded. Refer to the appropriate NPL Supplement for hardware and operating system-specific details.

References:

Native Operating System Files as Diskimage Files - Section 7.3.4 of the Programmer's Guide
Native Operating System Files as "Raw" Devices - Section 7.3.5 of the Programmer's Guide
Catalog Access - Section 7.3.8 of the Programmer's Guide

$MSG

General Form:

Form 1

```
$MSG = alpha-expression
```

Form 2

```
alpha-variable = $MSG
```

Discussion:

Form 1 of the $MSG statement is used to set the status of the system message which is displayed at terminals on line 0 whenever a RESET or CLEAR is performed at that terminal.

Form 2 can be used to determine the current status of the $MSG system variable.

NOTE: **Only terminals on the same processor share a single $MSG. Networked systems have separate $MSG values.**

Examples:

```
0010 $MSG = "SYSTEM SHUTDOWN 12:45 PM"
0010 $MSG = "*** INTERPRETIVE RUNTIME ***"
0010 A$=$MSG
0010 B$(2) = $MSG
```

Compatibility Issues:

Wang 2200 Basic-2 allows the $MSG system variable to be modified from terminal #1 only. NPL allows modification from any terminal on the system.

References:

Multi-user Functions - NPL Supplements

References:

Multi-user Functions - NPL Supplements

$NAMEOF() - Built-in String Function

General Form:

```
    $NAMEOF (identifier)
```

Where:

```
    identifier = { FUNCTION 'function_name[$]}
                 { PROCEDURE 'procedure_name }
                 { DEFFN' deffn_name         }
                 { FIELD .field_name[$][()]  }
                 { RECORD record_name        }
```

Discussion:

The $NAMEOF string functions are used to retrieve the name of public identifiers as string constants.

Use of the $NAMEOF string functions allow the developer to clearly identify string literals that refer to names which will be used indirectly (i.e., passed to another function to be used as callbacks).

Literals specified in this manner are automatically changed when a specified function name is changed using a corresponding RENAME function statement.

NPL requires (at resolve time) that the named function/ procedure/ field/ deffn must be previously declared as a PUBLIC item or an error is generated. This will avoid the common mistake of attempting to use the name of a non-PUBLIC item as an indirect reference.

Compiled code for a literal derived from the $NAMEOF string function is smaller than a literal containing the same name, if the name is 5 characters or longer.

$NAMEOF (cont.)

Examples:

```
0010  FUNCTION 'ScreenUpdate(A)/PUBLIC
    :  A=79
    :  RETURN(A)
    : END FUNCTION
0020  PRINT $NAMEOF(FUNCTION 'ScreenUpdate)

  :RUN

    ScreenUpdate

 :RENAME FUNCTION 'ScreenUpdate TO 'VideoUpdate

:LIST

0010  FUNCTION 'VideoUpdate(A)/PUBLIC
    :  A=79
    :  RETURN(A)
    : END FUNCTION
0020  PRINT $NAMEOF(FUNCTION 'VideoUpdate)
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

RENAME FUNCTION
RENAME PROCEDURE
RENAME DEFFN'
RENAME FIELD
RENAME RECORD

$NETID

General Form:

```
     alpha-receiver = $NETID
```

Discussion:

The $NETID function returns the full network physical station number on those operating environments that support this option. This is a displayable 12-hex digit number and may be used wherever string literals are legal.

Examples:

```
10 PRINT $NETID
```

Compatibility Issues:

This statement is only supported with Release IV or greater.

This statement returns meaningful values only on Novell NetWare installation.

References:

NEXT

General Form:

```
NEXT index-variable [,index-variable]...
```

Where:

```
index-variable = the numeric-scalar variable(s) which corre-
                 sponds to active FOR/NEXT loop(s).
```

Discussion:

The NEXT statement is used to indicate the end of a related structured FOR/TO/BEGIN or FOR/TO loop. One or more index-variables may be specified that must relate to one or more prior FOR/TO statements. Execution of a NEXT statement without having previously executed a related FOR/TO statement is diagnosed as a RunTime error.

If a positive STEP value is used, the index-variable is tested to determine if the index-variable (+ STEP value) exceeds the final loop value. If a negative STEP value is used, the index-variable is tested to determine if the index-variable (- STEP value) is less than the final loop value. In either case, if the test is true, program execution continues with the program statement following the NEXT statement and the increment (or decrement) is not performed. If not, the index-variable is incremented (or decremented) and execution continues with the statement following the corresponding FOR statement.

If the STEP value in a FOR/TO statement is 0, upon execution of the corresponding NEXT statement, the loop ends and program execution continues with the statement following the corresponding FOR statement.

If more than one index-variable is specified, the first is evaluated against its related FOR/TO loop as indicated above. If that loop is terminated, the next index-variable is evaluated. The process is repeated until an index-variable is found with a loop which does not terminate, or there are no more index-variables.

Examples:

```
:0010 FOR I=1 TO 10
:0020    FOR J=1 TO 50
:0030       FOR K=I TO J+10 STEP 2
:0200    NEXT K,J
:0300 NEXT I
```

NEXT (cont.)

Compatibility Issues:

References:

BREAK
FOR/TO
FOR/BEGIN

NEXT CLEAR

General Form:

```
NEXT CLEAR
```

NOTE: The use of this statement is not recommended. Use BREAK inside a FOR/BEGIN as a better alternative.

Discussion:

The NEXT CLEAR statement is used to perform the same function as the NEXT statement, except that the currently executing FOR/TO loop is terminated, regardless of the value of the index-variable. Execution continues with the statement following NEXT CLEAR.

NEXT CLEAR is valid only where NEXT would be valid. If a FOR/NEXT loop is not the top entry in the return stack, NEXT CLEAR results in a P40 (No Corresponding FOR for NEXT Statement) error.

Examples:

```
0010 NEXT CLEAR

:0010 A=1
:0020 FOR X=1 TO 10
:0030    PRINT X
:0040    A=A*X
:0050    PRINT X,A
:0060    IF A<50 THEN NEXT X<
:      ELSE DO<
:        NEXT CLEAR<
:        PRINT "VALUE OF A EXCEEDS 50 - FOR/NEXT LOOP TERMINATED"<
:        ENDDO
:0070 PRINT "NUMBER OF LOOPS COMPLETE = ";X

:RUN

    1              1
    2              2
    3              6
    4              24
T5  120
    VALUE OF A EXCEEDS 50 - FOR/NEXT LOOP TERMINATED EARLY
    NUMBER OF LOOPS COMPLETE = 5
```

NEXT CLEAR (cont.)

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.
NEXT CLEAR is not supported on the Wang 2200.

References:

FOR/TO
NEXT

NUM Function

General Form:

```
NUM(alpha-variable)
```

Discussion:

The NUM function returns the number of numeric characters in an alpha-variable. This is valid wherever a numeric-expression is legal.

Characters considered to be numeric are "0-9", ".", "+", "-", "E", and blank spaces. These characters must also be in a logical order to form a number. The NUM function returns the value up to the first non-numeric character.

NUM is particularly useful before numeric conversions to assure that only numeric characters are converted.

Examples:

```
0010 X = NUM(A$)
0010 IF NUM(W$) = 0 THEN GOTO 1000
0010 FOR I = 1 TO NUM(W$)

:0010 INPUT N$
    : IF NUM(N$)>=LEN(N$) THEN PRINT "NUMERIC"
    :    ELSE PRINT "NOT NUMERIC"
:RUN
? ABC
NOT NUMERIC
:RUN
? 49
NUMERIC

:0010 X$="21458#ID" : PRINT NUM(X$)
:RUN
5
```

Compatibility Issues:

References:

$NUMBERS

General Form:

Form 1

```
$NUMBERS = alpha-expression
```

Form 2

```
alpha-variable = $NUMBERS
```

Discussion:

Form 1 of the statement is used to set the status of the $NUMBERS system variable.

Form 2 can be used to determine the current status of the $NUMBERS system variable.

$NUMBERS is a one-byte system variable used in the interpretive RunTime to control generation of statement numbers in p-code program lines. If $NUMBERS contains a value of HEX(00) (the default) then statement numbers are not generated. If $NUMBERS contains a value of HEX(01) then statement numbers are generated.

The $NUMBERS statement is functionally equivalent to the NUMBERS option in the compiler (B2C). A value of HEX(00) is equivalent to NUMBERS OFF; a value of HEX(01) is equivalent to NUMBERS ON.

$NUMBERS affects the generation of p-code as program lines are entered using the line editor or processed by the $OBJECT function. It has no effect when programs are saved to disk.

Statement numbers are used on the error display of the non-interpretive RunTime to provide better error diagnostics. If statement numbers are present, the non-interpretive RunTime displays the statement number along with the program line number when an error occurs. Otherwise, an offset in hexadecimal byte location from the start of the program line is displayed.

$NUMBERS (cont.)

Statement numbers, if generated, occupy two bytes per statement in the p-code file.

$NUMBERS performs no operation (NOP) on the non-interpretive RunTime (RTP).

Examples:

```
0010 X$=$NUMBERS
0010 $NUMBERS=BIN(0)
0010 $NUMBERS=HEX(01)
```

Compatibility Issues:

The $NUMBERS statement is not valid in Wang 2200 Basic-2.
This statement is supported only with Release 2.0 or greater

References:

$OBJECT
NUMBERS Option - Section 14.13 of the NPL Programmer's Guide

$OBJECT

General Form:

```
receiver-variable1 = $OBJECT(alpha-variable2 [,alpha- varible3])
```

Where:

```
alpha-variable2 = ASCII text to be converted to p-code.


alpha-variable3 = a long identifier table, if long
                  identifiers are to be used.
```

Discussion:

The $OBJECT function is used to generate NPL compatible p-code from single lines of ASCII program text. $OBJECT is useful for the dynamic generation of NPL programs.

$OBJECT is executable only in the interpretive version of the RunTime (RTI). The syntax is supported in the non-interpretive version (RTP) but no operation takes place.

Input to $OBJECT (alpha-variable2) must be ASCII text as it would normally be submitted to the line editor of the interpreter. If the source code may contain long identifier names, a long identifier table (alpha-variable3) must be specified.

NOTE: The p-code generated by $OBJECT is affected by the status of $KEEPREMS and $NUMBERS just as is code entered from the editor. Refer to $KEEPREMS and $NUMBERS for details.

The output produced is the single line record as it appears in the p-code files, followed by 5 bytes of status information, which reflects the syntactical correctness of the line. Bytes (1) one and (2) two of the status information, contain (low byte first) a pointer to the byte from the alpha-variable which was being processed when the syntax error was generated. Bytes (3) three and (4) four of the status information contain (low byte first) the error code of the error produced. Byte 5 of the status unformation contains a binary count of the number of errors in the line. A value of HEX(00) indicates no syntax errors.

Any unused portion before the last 5 bytes of the receiver-variable status information, is filled with spaces (HEX(20)).

$OBJECT (cont.)

The output from $OBJECT is limited by the size of internal buffers used by the Run-Time. This buffer size is at least 512 bytes and may be larger on some hardware versions. In cases where the internal buffer size is exceeded, the last 5 bytes returned still contain the status information described above and an error is indicated in the error status bytes.

If new long identifiers are present, they are added to the specified long identifier table.

Practical use of $OBJECT requires the assembly of a p-code file from its component p-code lines after use of $OBJECT, with the addition of a suitable label, end of file mark and long identifier table. As of Release IV it is no longer possible for the developer to directly manipulate p-code files directly. As such, Niakwa has provided a set of Library routines to help manipulate these files. Please refer to Chapter 3 of the Statements Guide for information on these Library files and an example of the use of $OBJECT and $SOURCE. Also refer to Chapter 5 of the Programmers Guide for more details on $SOURCE and $OBJECT.

STOP *WARNING--If the label is 2.00.0x or higher, long identifier names require use of the library functions to decompile. Refer to Section 3.2 for an example of the use of $OBJECT.*

NOTE: Refer to Chapter 3 for further information.

Examples:

```
0010 PcodeLine$=$OBJECT(SourceLine$,IdentifierTable$)
```

Compatibility Issues:

The alpha-variable3 option is only available with Release IV or later, and is required to use long identifiers.

This statement is not valid in Wang 2200 Basic-2.

This statement requires Revision 2.00 or higher of the Interpreter.

References:

$SOURCE
$KEEPREMS
$NUMBERS
Chaper 5 Programmers Guide
Chaper 3 Statements Guide

ON ERROR

General Form:

```
ON ERROR alpha-variable1, alpha-variable2 GOTO line-number
```

Discussion:

The ON ERROR statement captures recoverable and non-recoverable errors (error codes S10 and above, excluding any computational errors suppressed by SELECT ERROR).

When an error occurs which is captured by the ON ERROR statement, the following sequence of events occurs:

- RETURN information for GOSUB and GOSUB' and FOR/TO information since the last FUNCTION or PROCEDURE call is cleared from the system stacks. Consequently, subroutines and loops which were in progress when the error occurred may not normally be resumed.
- Alpha-variable1 is assigned the value of the error code which occurred. The value is a 3-character code. For errors in the range of 1-99, the three-character code consists of a one-byte error class followed by the two-byte error code. For error codes of 100 or greater, the three-character code is the error code with no preceding error class specification.
- Alpha-variable2 is assigned the value of the line number on which the error occurred. The value assigned is a 5-character displayable format with leading zeroes added if the line number is less than 1000 (e.g., "0600").
- Program execution continues at the start of the line-number specified in the ON ERROR statement.

The ON ERROR statement may not appear inside the body of a FUNCTION or a PROCEDURE.

ON ERROR (cont.)

Use of the ON ERROR statement is discouraged in modules that define FUNCTION's, PROCEDURES or PUBLIC DEFFNs, since execution of the statement causes an unstructured exit from any exiting FUNCTION or PROCEDURE body, and loses the return information that is required to return to the caller of a DEFFN'.

Where possible, the ERROR statement should be used instead to catch specific errors.

ON ERROR may be used to catch non-recoverable errors (especially programmer errors such as invalid subscripts) which should not be occurring in an operational program. The program may need to perform cleanup work (if the problem has occurred in a critical section of the program) before advising the operator that an error has occurred.

NOTE: For most error conditions, use of the ERROR statement is preferred because:

- **After processing errors which occur during the execution of a subroutine or FOR/TO loop, it may be necessary to resume execution of the subroutine or loop after recovering from the error. This is possible with the ERROR statement, but it is not possible if the error is captured by an ON ERROR statement.**

- **Error recovery procedures may more easily be written to handle specific error conditions which occur on a given statement (e.g., PACK statements would check for numbers which are too large, etc.). The program text which is branched to by the ON ERROR may only do specific corrections based on the value in the two alpha-variables. Any conditions which compare the line number value in alpha-variable2 may become incorrect if the program lines are moved or renumbered.**

- **Because the ERROR statement appears next to the statement with the possible error condition, if revisions are made to that statement it is obvious to the programmer that consideration should be given to error handling. By contrast, an ON ERROR statement may be located far away from the statements whose errors it is intended to capture.**

ON ERROR (cont.)

Examples:

```
0010 ON ERROR A$,B$ GOTO 1000
0010 ON ERROR X1$,X2$ GOTO 100

:0010 ON ERROR A$,B$ GOTO 100
:0020 DIM X$256
:0030 DATA LOAD BA T (30000) X$
    : REM No such sector number
    : STOP
:0100 PRINT "Error ";A$;" on Line ";B$

:RUN
Error I98 on Line 0020
```

Additional information about errors may be obtained under program control by use of the ERR$ and $OSERR statements.

Compatibility Issues:

References:

ERROR
SELECT ERROR
ERR$
$OSERR
NPL Error Codes - Appendix B of the Programmer's Guide

ON/GOSUB

General Form:

```
ON numeric-expression GOSUB [[target],]...target
                 [:ELSE {simple-statement        }]
                        {DO[:statement]...:ENDDO }]
```

Where:

```
target = {line-number     }
         {statement-label }
```

NOTE: The use of this statement is not recommended. Refer to SWITCH as a better alternative.

Discussion:

The ON/GOSUB statement is used to execute one of several possible subroutines based on the integer portion of the value of the numeric-expression.

When the ON/GOSUB statement is executed, program control is branched to the corresponding subroutine in the GOSUB list. A subroutine may be specified as either a line number or a statement label. A null entry is specified in the GOSUB list by consecutive commas. A null entry specifies that no subroutine branch is to be executed for that particular value in the GOSUB list. For example, if the numeric-expression in an ON/GOSUB statement were equal to 4, program execution would branch to the subroutine specified by the fourth line-number in the ON/GOSUB list. If a fourth subroutine does not exist (a null entry or fewer than 4 subroutines in the list), a subroutine call is not performed and program execution continues with the next program statement following the ON/GOSUB statement.

If the target line-number or statement-label is located in a function body, this statement must also be in the body of the same function.

If the ON/GOSUB statement is located inside a function body, all target line-numbers and statement-labels must also be located in the body of the same function.

If the value of the numeric-expression is less than or equal to zero, a subroutine call is not performed and program execution continues with the next program statement following the ON/GOSUB statement.

ON/GOSUB (cont.)

The statement or DO group following the optional ELSE clause is only executed if a subroutine call is not executed. That is, if the value of the numeric-expression points to a null entry (or non-existing entry) in the GOSUB list, the ELSE clause is executed.

Examples:

```
0010 ON X GOSUB Code_1,Code_2,Code_3,Code_4,,Code_6,Code_7
0020 ON Button+1 GOSUB OK,Cancel,Retry,9999:ELSE STOP

0010 ON X+1 GOSUB 100,200,300,400
0010 ON I*.333 GOSUB 1000,,1200
0010 ON VAL(STR(V$(),34)) GOSUB ,,,1000,2000,1000,2000

:0010 INPUT A
:0020 ON A GOSUB 100,200,300,,,600,900
    :    ELSE GOSUB 1000
:0030 GOTO 2000: REM DONE
:0100 PRINT "LINE 100": RETURN
:0200 PRINT "LINE 200": RETURN
:0300 PRINT "LINE 300": RETURN
:0600 PRINT "LINE 600": RETURN
:0900 PRINT "LINE 900": RETURN
:1000 PRINT "NO LISTED SUBROUTINE": RETURN
:2000 REM DONE

:RUN
? 2
LINE 200

:RUN
? 10
NO LISTED SUBROUTINE

:RUN
? 5
NO LISTED SUBROUTINE
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

Statement labels are supported only on NPL Revision 4.0 or greater.

References:

DO/ENDDO
ELSE

ON/GOTO

General Form:

```
ON numeric-expression GOTO [[target],]...target
                 [:ELSE {simple-statement        }]
                        {DO[:statement]...:ENDDO }]
```

Where:

```
target = {line-number     }
         {statement-label }
```

NOTE: The use of this statement is not recommended. Refer to SWITCH as a better alternative.

Discussion:

The ON/GOTO statement is used to transfer program execution to one of several possible lines based on the integer portion of the value of the numeric-expression.

When the ON/GOTO statement is executed, program control is transferred to the corresponding line-number or statement-label in the ON/GOTO list.

If the target line-number or statement-label is located in a function body, this statement must also be in the body of the same function.

If the ON/GOTO statement is located inside a function body, all target line-numbers and statement-labels must also be located in the body of the same function.

If the numeric-expression is less than or equal to zero, a transfer is not performed and program execution continues with the next program statement following the ON/GOTO statement.

Examples:

```
0010 ON X GOTO Case_1,Case_2,Case_3,Case_4,,Case_6,Case_7
0020 ON Button+1 GOTO OK,Cancel,Retry,9999:ELSE STOP

0010 ON X+1 GOTO 100,200,300,400
0010 ON I*.333 GOTO 10000,,12000
0010 ON VAL(T$) GOTO ,,,,1000,2000,1000,2000
0010 ON VAL(STR(A$,12) GOTO ,,100,200,,9999
```

ON/GOTO (cont.)

```
:0010 INPUT A
:0020 ON A GOTO ,,100,200,300,,,600,900
:0030 PRINT "NO TRANSFER": GOTO 1000
:0100 PRINT "line 100": GOTO 1000
:0200 PRINT "line 200": GOTO 1000
:0300 PRINT "line 300": GOTO 1000
:0600 PRINT "line 600": GOTO 1000
:0900 PRINT "line 900": GOTO 1000
:1000 REM DONE

RUN
? 1
NO TRANSFER

:RUN
? 3
line 100

:RUN
? 10
NO TRANSFER

:RUN
? 4
line 200
```

Compatibility Issues:

Statement labels are supported only in NPL Revision 4.0 or greater.

References:

ON/SELECT

General Form:

```
ON numeric-expression SELECT [[item-list];]...item-list
               [:ELSE {statement                 }]
                      {DO [:statement] ...: ENDDO}
```

Where:

```
item-list   = select-item [,select-item]...


select-item = as defined in SELECT statement general form.
```

NOTE: The use of this statement is not recommended. Refer to SWITCH as a better alternative.

Discussion:

The ON/SELECT statement is used to conditionally assign a list of device-addresses or other SELECT parameters to corresponding internal device table entries, based on the integer portion of the value of the numeric-expression.

One or more devices may be selected on a given numeric-expression by assigning them to the same item-list (separated by comma (,)).

When the ON/SELECT statement is executed, select-items within an item-list are assigned based on the value of the numeric-expression. A null item-list is specified by consecutive semicolons. A null item-list specifies that no assignment is to be executed for that particular value of numeric-expression. For example, if the numeric-expression in an ON/SELECT statement were equal to 4, select-items in the fourth item-list of the ON/SELECT statement would be assigned. If a fourth item-list does not exist (a null entry or less than 4 item-lists in the ON/SELECT statement), a SELECT operation is not performed.

If the numeric-expression is less than or equal to zero, a SELECT operation is not performed.

ON/SELECT (cont.)

The statement or Do Group which follows the optional ELSE clause is only executed if no item-list of select-items is assigned. That is, if the value of the numeric-expression points to a null item-list (or non-existing item-list) in the statement, the ELSE clause is executed.

Examples:

```
0010 ON I SELECT DISK/D10,PRINT/215,LIST/215;PRINT/005,LIST/005
```

If INT(I)= 1, then disk address D10, printer address 215, and list output address 215 would all be selected. If INT(I)= 2, print address 005, and list address 005 would be selected. If I or I= 3, then no SELECT is performed.

```
0010 ON VAL(A$) SELECT DISK/D10; DISK/D11,PRINT 217;; DISK/D12;
     DISK/D13
     : ELSE SELECT DISK/D14
```

If VAL(A$)= 1 disk address D10 is selected.
If VAL(A$)= 2 disk address D11 and PRINT address 217 are selected.
If VAL(A$)= 4 disk address D12 is selected.
If VAL(A$)= 5 disk address D13 is selected.
If VAL(A$) has any other value, disk address D14 is selected (in the ELSE clause).

```
0010 ON X SELECT D;R;G
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

References:

Internal Device Table - Section 7.2.3 of the Programmer’s Guide
Printer Devices - Section 7.7 of the Programmer’s Guide
DO/ENDDO
ELSE
SELECT

$OPEN

General Form:

```
$OPEN [line-number,]{file-number   } [,file-number   ]...
                    {device-address} [,device-address]
                    {<address-var> } [,<address-var> ]
```

Discussion:

The $OPEN statement is used to reserve a diskimage or printer for exclusive access so that it cannot be accessed by any other NPL process on the system.

$OPEN is typically used with diskimages to ensure that critical updates are performed to completion without intervening access from other NPL processes. $OPEN is also used with PRINT type devices to ensure that output is not intermingled with output from another NPL process during printing of a report.

If any of the devices in the list are already reserved for exclusive access by another NPL process, the system either: branches to the specified line-number with no devices in the list being reserved or, if no line-number is specified, waits until the device becomes available.

A device remains reserved for exclusive access until the execution of a $CLOSE, END, or $END statement.

NOTE: Pressing the HELP key causes all exclusive access to be relinquished. When HELP is then exited, exclusive access is not automatically regained.

Examples:

```
0010 $OPEN #A,#B
0010 $OPEN /D11, /D23, /D24
0010 $OPEN 100, #2
0010 $OPEN 8010, #C
0010 $OPEN /215
0010 $OPEN <A$>
```

$OPEN (cont.)

Compatibility Issues:

In a multi-user environment, the $OPEN statement emulates the Wang 2200 except in the case of disk devices. In this case, $OPEN on a disk address is DISKIMAGE-specific, not DEVICE-specific.

For example, on a Wang 2200, assuming D20 is the removable platter on a Wang 2200 phoenix drive, $OPEN /D20 "hogs" the entire device (platters /D20 - /D25). However, under NPL, $OPEN /D20 "hogs" only the diskimage equivalent to /D20.

References:

$CLOSE
NPL Diskimages - Section 7.3.1 of the Programmer’s Guide
Printer Devices - Section 7.7 of the Programmer’s Guide
Exclusive Access - Section 7.2.4 of the Programmer’s Guide

$OPTIONS

General Form:

Form 1

```
$OPTIONS = alpha-expression
```

Form 2

```
alpha-receiver = $OPTIONS
```

Where:

```
alpha-receiver = a minimum of 64 characters.
```

Discussion:

Form 1

Form 1 of the $OPTIONS statement allows the NPL application program to modify the contents of the $OPTIONS system variable.

Form 2

Form 2 allows examination of the current status of the $OPTIONS system variable.

$OPTIONS is a 64-byte system variable containing option selection values that allows NPL to take advantage of various native operating environment features and screen handling routines, where applicable. Specifically it contains:

Byte 1:	Replacement attribute for the underline character to be used on color monitors by the IBM version of the RunTime Program.	
	HEX(1F) Default	Bright white on blue background.
Byte 2:	Switch to determine whether or not keyboard status information (CAPS LOCK and NUM LOCK indicators) is to be displayed on line 25 of the monitor (IBM version only).	
	HEX(00) Default	Disables the status display.
	HEX(01)	Enables the status display.

Byte 3:	Switch for "noise" suppression option on IBM color monitors.	
	HEX(00) Default	Indicates that no "noise" suppression is to take place.
	HEX(01)	Indicates that noise suppression is to take place.
Byte 4:	Replacement character for the "$" character in PRINTUSING image statements. When a "$" is specified in a PRINTUSING image as a currency sign, this value appears on output.	
	HEX(24) Default	Prints a "$".
Byte 5:	Replacement character for the "," (comma) character in PRINTUSING image statements. When a comma is specified in a PRINTUSING image as a digit separator, this value appears on output.	
	HEX(2C) Default	Prints a ",".
Byte 6:	Replacement character for the "." (decimal point) character in PRINTUSING image statements. When a "." is specified as a decimal point in a PRINTUSING image, this value appears on output.	
	HEX(2E) Default	Prints a ".". The "$", ","(comma), and "." decimal point replacements are used primarily for foreign currency applications.
Byte 7:	Keyboard translation complex key lead value. When this value is received by the keyboard handlers from the native operating system, a complex key sequence is assumed to follow.	
Byte 8:	ANSI output mode flag. Defines terminal functionality type. Values of this byte have the same meaning as byte 9 of $MACHINE. Refer to the NPL Supplements for details on acceptable values for the operating system in use.	
Byte 9:	Keyboard translation complex key trailing value. After receiving a complex key sequence from the native operating system, if this byte is not HEX(00), the keyboard handlers wait for a trailing code to finish the sequence.	
	HEX(00) Default	
Byte 10:	Alternate keyboard translation complex key lead value. When this alternate value is received by the keyboard handlers from the native operating system, a complex key sequence is assumed to follow.	
	HEX(00) Default	No alternate code is allowed.
Byte 11:	Alternate keyboard translation complex key trailing value. After receiving an alternate complex key sequence from the native operating system, if this byte is not HEX(00), the keyboard handlers wait for a trailing code to finish the sequence.	
	HEX(00) Default	
Byte 12:	Switch for suppression of the "Reset", "Step", and "Continue" options from the HELP display.	

	HEX(00) Default	Options enabled.
	HEX(01)	Suppresses these options from the HELP display. Resetting byte 12 to HEX(00) reenables these options on the HELP display.
Byte 13:	Switch for suppression of the "HALT" key.	
	HEX(00) Default	Key enabled.
	HEX(01)	Suppresses the "HALT" key. Resetting byte 13 to HEX(00) reenables the "HALT" key.
Byte 14:	Perform implicit $BREAK after disk I/O.	
	HEX(00) Default	Indicates that no break is to take place.
	HEX(01)	Indicates that the remaining timeslice for the task is released after each disk I/O statement. This feature is used to improve performance on some operating systems. Refer to the NPL Supplements for details.
Byte 15:	Alternate font selection on VT100/VT200 and Altos 3 terminals. Refer to Appendix D of the Programmer's Guide for acceptable values for specific terminals.	
	HEX(41) Default	Hexcode for A--use BRITISH character set
Byte 16:	Not all math operations use the co-processor. Refer to Chapter 5 of the NPL Supplements for details on the availability and functionality of the math co-processor on specific hardware versions of NPL.	
	HEX(00) Default	Indicates that the math co-processor should not be used for any NPL math operations.
	HEX(01)	Indicates that the math co-processor should be used when available for NPL math operations.
Byte 17:	132-column screen support. Refer to Section 7.3.23 of the Programmer's Guide for further details on 132-column mode.	
	HEX(00) Default	Indicates that dynamic switching between 80-column mode and 132 column mode is not enabled.
	HEX(01)	Indicates that this feature is enabled.
Byte 18/19	Extended range for LINPUT/INPUT. Byte 18 defines the low value of the extended range. Byte 19 defines the high value of the extended range. Refer to Section 7.5.5 of the Programmer's Guide for further details on extended range for LINPUT/INPUT.	
	HEX(00) Default	In both bytes

Byte 20:	"Raw" output in remote mode. "Raw" output mode is applicable only on PC monitors under MS-DOS when the non-interpretive RunTime is operating in "remote" mode. Remote mode is used when the "R" startup option is specified	
	HEX(00) Default	Indicates that "raw" output mode should not be used.
	HEX(01)	Indicates that "raw" output mode should be used.
Byte 21:	Controls display of "bright" attribute on terminals where "normal" is actually "dim". Refer to Appendix D of the Programmer's Guide for information on specific terminals where this feature is supported.	
	HEX(00) Default	Indicates that "bright" is displayed as "bright" and "normal" is displayed as "dim".
	HEX(01)	Indicates that "bright" is displayed as "dim" and "normal" is displayed as "bright".
Byte 22:	Provides power-on default background/foreground color selection for supported color terminals. Refer to Section 7.3.18 of the Programmer's Guide for details.	
	HEX(00) Default	
Byte 23:	Provides power-on default perimeter/underline color selection for supported color terminals. Refer to Section 7.3.18 of the Programmer's Guide for details.	
	HEX(00) Default	
Byte 24:	Controls attribute display on IBM PC monitors in graphics (/G) mode. Refer to Chapter 6 of the DOS Supplement for details.	
Byte 25:	Color replacement value for no bright, no blink in EGA graphics mode.	
	HEX(07) Default	Indicates white on black. Refer to Chapter 6 of the DOS Supplement for details.
Byte 26:	Color replacement value for bright, no blink in EGA graphics mode. Refer to Chapter 6 of the DOS Supplement for details.	
	HEX(01) Default	Indicates bright white on black.
Byte 27:	Color replacement value for no bright	
	HEX(04) Default	
Byte 28:	Color replacement value for bright	
	HEX(14) Default	
Byte 29:	Allows for remapping of available colors for video attribute replacement in graphics mode on terminals with less than 64K that have only four colors. Refer to Chapter 6 of the appropriate NPL Supplement.	
	HEX(xy)	Where x replaces color for black and y replaces blue.
	HEX(0F) Default	Replaces blue with bright white.

Byte 30:	Allows for remapping of available colors for video attribute replacement in graphics mode on terminals with less than 64K that have only four colors.	
	HEX(xy)	Where x replaces red and y replaces white.
	HEX47) Default	
Byte 31:	Controls certain features for terminal emulators which do not provide 100% support of the terminal being emulated. This feature is provided strictly as a convenience for those users who must use emulation products. These emulation products are not supported for use with NPL. Each defined bit can be used to suppress a NPL feature for the terminal in use. Defined bits are:	
	HEX(00) Default	No features supplied.
	HEX(01)	The HELP display highlights only the "current" field (which would normally appear as inverse video). Other fields which would display as bright are not highlighted. This feature is useful on terminal emulators which do not support multiple video modes. Only the HELP display is affected by this option. Programs which use multiple attributes may not display well on these terminals.
	HEX(02)	The terminal has no local printer capability. If this bit is set, printers with the LCL=Y $DEVICE clause are not supported and attempting to access such printers yields a P48 error (Illegal Device Specification).
	HEX(04)	132-column mode is not supported. If this bit is set, the dynamic screen size option is ignored and the current width is not changed. Other bits are reserved and should not be modified.

Byte 32:	Cursor style selection. For terminals where the cursor style must be set under program control, this byte may be used to instruct the non-interpretive RunTime which cursor style to set. Possible values:	
	HEX(00) Default	Use current cursor style or built in defaults.
	HEX(01)	Set cursor style to line.
	HEX(02)	Set cursor style to block. When a non-zero value is specified, the cursor style is set permanently to the specified style and is not reset when the non-interpretive RunTime is exited. The cursor style is changed on the first cursor on (HEX(05)) or cursor blink (HEX(02050E)) sequence sent to the terminal. Refer to Appendix D of the Programmer's Guide for details on the effect of this option on specific terminals.
Byte 33:	Byte 33 of $OPTIONS is used to determine the availability of the Native Operating System, Kill NPL, Enable/Disable Keyboard Logging, and Diag options on the HELP processor. The options are controlled as follows:	
	HEX(00) Default	Indicates that all options are displayed.
	HEX(01) bit - 0	Display Native Operating System option
	HEX(01) bit - 1	Suppress Native Operating System option
	HEX(02) bit - 0	Display Kill NPL option
	HEX(02) bit - 1	Suppress Kill NPL option
	HEX(04) bit - 0	Display Enable/Disable Keyboard Logging options
	HEX(04) bit - 1	Suppress Enable/Disable Keyboard Logging options
	HEX(08) bit - 0	Display Diag option
	HEX(08) bit - 1	Suppress Diag option
	Other bits reserved and must be zero.	

Programmers are advised to preserve the reserved bits by using a logical operation such as OR to assign values to this byte.

For example:

```
0010 DIM O$64
0020 O$=$OPTIONS
0030 STR(O$,33,1)=STR(O$,33,1) OR HEX(05):REM Suppress all three
0040 $OPTIONS=O$
```

NOTE: If the Kill NPL option is suppressed, it may be necessary to kill the task from another terminal or reboot the system if certain fatal conditions are encountered during RunTime execution. An example of this would be an error occurring during execution of the non-interpretive RunTime.

$OPTIONS (cont.)

Also, if the Diag option is suppressed, Enable/Disable Keyboard Logging is implicitly suppressed regardless of the value of the HEX(04) bit since these options appear only on the Diag screen.

Refer to SELECT LOG for further details on Keyboard Logging.

Refer to Chapter 11 of the Programmer's Guide, for further details on the Help Processor display.

Byte 34:	Value of #PI	
	HEX(00) Default	Indicates that the value assigned to #PI should be 3.1415926535898. This is the most accurate value possible in NPL.
	HEX(01)	Indicates that the value assigned to #PI should be 3.141592653590. This is the value that was returned on revisions of NPL prior to revision 2.01. This value should typically only be used by applications that compare #PI to values that may have been stored using older versions of NPL.
	Other values are reserved and should not be set by the application. Regardless of the value of byte 34, the default value for #PI is still used for all transcendental functions that deal with radians.	
Byte 35:	To apply either Wang logic or Niakwa logic to the $OPEN operation.	

	HEX(00) Default	When $OPEN encounters a device "hogged" by another task, devices lower on the list remain hogged by the task issuing the $OPEN. This logic was used by all releases prior to 3.01.
	HEX(01)	When $OPEN encounters a device "hogged" by another task, devices lower on the list are released, even if they were previously hogged by an earlier $OPEN statement. This is consistent with Wang 2200 Basic-2.
Byte 36:	Indicates the number of $BREAK units released is to be accumulated and that a UNIX timeslice should occur when the number of units accumulated is equal to or greater than the binary value specified in this byte.	
	HEX(00) Default	No UNIX timeslice is ever released.
Byte 37:	Specifies if the characters input as response to LINPUT and INPUT statements are echoed by NPL. If set to HEX(01) and the UNIX echo parameter is set OFF before entering the Niakwa RunTime, no characters are echoed to the screen when keys are pressed in response to the LINPUT or INPUT statements. If echo is on before entering the Niakwa RunTime, then the standard LINPUT or INPUT is used despite the value of this byte. When this byte is set to any value other than HEX(01), the standard LINPUT or INPUT is used despite the status of UNIX echo parameter. No other NPL statements are modified. The operation of KEYIN is not affected by this byte. HEX(00)-no suppression takes place.	
	HEX(00) Default	Standard input.
	HEX(01)	No characters echoed.
	By using the status command on the echo parameter, the developer is able to avoid complex logic in determining whether to set byte 37 of $OPTIONS on sites where some terminals are local and should echo characters normally while others are operating remotely and the echoing should be suppressed. The normal setting for the echo parameter is ON in UNIX environments.	
Byte 38:	Used to require all numeric and alpha-scalar variables to be explicitly dimensioned prior to first reference.	
	HEX(00) Default	Numeric and alpha-scalar variables are implicitly dimensioned upon first reference
	HEX(01)	Numeric and alpha-scalar variables must be explicitly dimensioned prior to first reference. NPL flags all undeclared variables with a P55 error (Undefined Variable).

	If the default value (HEX(00)) is used, a variable may be declared implicitly, depending on the context in which the first variable reference in the program appears, according to the following table: Location of First Reference — Default Allocation Type Within a function body — Not legal; error occurs Outside all function bodies — DIM/STATIC NOTE: Constant variables must always be explicitly declared.	
Byte 39:	Used to specify that no implicit $OPEN is to be performed on DATA LOAD BA and DATA LOAD BM when the platter has not been explicitly hogged by any $OPEN.	
	HEX(00) Default	Implicit $OPENs are to occur as they have in prior releases.
	HEX(01)	Implicit $OPENs are suppressed.
Byte 40:	Specify whether bytes 4 to 35 of the file trailer sector are used by the NPL to store a date and time stamp, filename, and other information for data files.	
	HEX(00) Default	NPL stores this information as done in previous releases.
	HEX(01)	
Byte 41:	Specify that $SOURCE should retain soft carriage returns embedded in the object code being translated to ASCII.	
	HEX(00) Default	Soft carriage returns are not to be retained as done in previous releases.
	HEX(01)	Soft carriage returns are retained

Byte 42	Used to modify the interpretive Runtime's behavior when unhandled program errors are encountered and immediate mode is entered (RTI only):		
	Bit Position/ Bit=1		
	HEX(00) Default	Same behavior as older versions (prior to NPL Revision 4.0). Beeps and $DEMO scripts keep running.	
	HEX(01)	Suppresses beep when an error message is displayed.	
	HEX(02)	Stops any running $DEMO script if an error occurs.	
	Other bits	Reserved; should always be set to zero (0).	
Byte 43	Specifies that RunTime will not yield to another Windows task while an $OPEN is active.		
	HEX(00) Default	Yield to other Windows tasks even if $OPEN's are outstanding.	
	HEX(01)	Yield to other Windows tasks only if no $OPEN's to network files have been granted.	
Byte 44	Used to control the startup editing characteristics of LINPUT/INPUT statements and immediate mode and program lines, and the characteristics of the "insert" key. The following defines each bit position's startup mode of operation.		
	BIT Position	BIT = 0	BIT = 1
	Immediate Mode Commands and Program Lines		
	HEX(01)	Overstrike Mode	Insert Mode
	HEX(02)	Insert key = insert char	Insert key toggles mode
	LINPUT Statements		
	HEX(04)	Overstrike Mode	Insert Mode
	HEX(08)	Insert key = insert char	Insert key toggles mode
	INPUT Statements		
	HEX(10)	Overstrike Mode	Insert Mode
	HEX(20)	Insert key = insert char	Insert key toggles mode
	RESERVED bits		
	HEX(40)	Should always be 0	
	HEX(80)	Should always be 0	
Byte 45	Suppresses or replaces the statement separators (":") at the start of multi-statement lines.		

	HEX(00) Default	Displays the statement separator (":") at the start of multi-statement lines.
	HEX(01)	Suppress when recalling lines for EDIT.
	HEX(02)	Suppress when displaying text in LIST (except LIST D).
	HEX(04)	Suppress when displaying text in LIST D.
	HEX(08)	Suppress when displaying source via $SOURCE function.
	HEX(10)	Display space instead of retrun graphic when editing lines.
Bytes 46-64:	Currently undefined.. Reserved for future use.	

When the $OPTIONS system variable is modified, specified options become effective immediately. They remain in effect until further modifications are made or until the current session is ended. Modifications are not carried over from one RunTime session to the next. At each execution of the RunTime program, the $OPTIONS variable is set to the specified default values.

Examples:

```
0010 DIM X$64
0020 X$=$OPTIONS          : REM GET CURRENT VALUES
0030 STR(X$,1,1)=HEX(4F)   : REM SET UNDERLINE OPTION TO BRIGHT
                            WHITE ON RED BACKGROUND
0040 STR(X$,2,1)=HEX(00)   : REM SET KEYBOARD STATUS DISPLAY OFF
0050 STR(X$,3,1)=HEX(01)   : REM SET 'NOISE' SUPPRESSION ON
0060 $OPTIONS=X$           : REM PUT OPTIONS INTO EFFECT
```

NOTE: **Programs which modify $OPTIONS should always fetch the current contents into a work variable, modify the work variable, and then reset $OPTIONS. The work variable should always be dimensioned to 64 bytes. This prevents problems in the future, should additional bytes of $OPTIONS be implemented.**

In many cases, it may be desirable to modify $OPTIONS based upon what machine is being used and what revision of the RunTime Program is being used. Refer to $MACHINE system variable and the appropriate NPL Supplement for details.

Compatibility Issues:

This statement is not valid in Wang 2200 Basic-2.

Refer to NPL Supplement(s) for hardware-specific details concerning the $OPTIONS statement.

Bytes 14-30 are valid only in NPL Revision 2.01 or greater.

Bytes 31-34 are valid only in NPL Revision 3.00 or greater.

$OPTIONS (cont.)

Bytes 35-41 are valid only in NPL Revision 3.20 or greater.

Bytes 42-45 are valid only in NPL Revision 4.0 or greater.

References:

OR Alpha-operator

General Form:

```
     alpha-receiver = [...] OR alpha-operand [...]
```

Where:

```
     alpha-operand = {literal-string   }
                     {alpha-variable   }
                     {ALL function     }
                     {BIN function     }
                     {system-variable  }
```

Discussion:

The OR logical alpha-operator performs a logical OR operation on the alpha-operand and the contents of the alpha-receiver, the result of which is then assigned to the alpha-receiver. The OR alpha-operator is legal only in an alpha-expression in an alpha-assignment statement.

The OR operation is performed on a byte-by-byte basis, moving from left to right in each field, for a number of bytes equal to the shorter of;

- The defined length of the alpha-receiver.
- The defined length of the alpha-operand (if the alpha-operand is an alpha-variable or system-variable, trailing spaces are included in the operation).

If the defined length of the alpha-operand is shorter than the defined length of the alpha-receiver, then the remaining bytes of the alpha-receiver remains unchanged (i.e., padding with spaces is not performed).

NOTE: With regard to the "OR" syntactic unit, this may also appear in conditional-expressions. However, the similarity is syntactical only and its use in a conditional-expression has a completely different meaning.

OR Alpha-operator (cont.)

Examples:

```
0010 STR((A$,4,5) = OR B$
0010 A$=C$ OR "0"
0010 B$=OR ALL(10)
0010 STR(A$,4,5)=OR B$

:0010 DIM A$5
:0020 A$="AbCdE"
:0030 A$=OR ALL(60)
:0040 PRINT A$
:RUN
abcde
```

Compatibility Issues:

References:

$OSERR

General Form:

```
alpha-receiver = $OSERR
```

Discussion:

The $OSERR function returns a string of text which contains the most recent native operating system error code and message received when a NPL error has occurred. $OSERR can not appear on the left side of an equivalence statement.

The text returned is comprised of two parts:

1. The native operating system error code.

2. A descriptive message based on this code. The descriptive message is obtained from the file RTIxERR.HLP (where x is a single letter representing the host operating system--"S" for SuperDOS for example). If the correct file for the operating system in use, along with the associated RTIxERR.IDX file, is not present or cannot be accessed, a descriptive message is not be returned by $OSERR.

NOTE: The RTIxERR.HLP file may be modified. If modified, it is necessary to execute the Indexed Help File Creation utility to create the associated RTIxERR.IDX file before the modified RTIxERR.HLP file can be properly accessed. Refer to Chapter 8 of the Programmer's Guide, for further information on indexed help files. Refer to the NPL Supplement for further information on the name and required location of RTIxERR files for the operating system.

The $OSERR value is returned even if the NPL error is trapped by ERROR on ON ERROR.

Typically, native operating system errors are generated only on I/O operations. If the NPL error was for a non-I/O related operation (X75 (Illegal Number) for example), $OSERR is typically blank.

$OSERR (cont.)

$OSERR is intended to be used only for informational purposes. Programmers are advised against attempting to base any logic decisions on the values returned by $OSERR. These values vary widely among different operating systems and may even vary widely on different revisions of the same operating system.

Examples:

```
0010 $FORMAT DISK T/D10,
   :ERROR E=ERR
   :E$=ERR$(E)
   :PRINT "Error ";&E;" - ";E$;" occurred"
   :N$=$OSERR
   :IF N$<>" " THEN PRINT "Native operating system error - ";N$
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

This statement is not supported on the Wang 2200.

References:

ERR
ERROR
ERR$
Chapter 8 of the Programmer's Guide - NPL Error Codes

PACK

General Form:

```
     PACK (image) alpha-variable FROM

          {numeric-array     } [,{numeric-array     }]...
          {numeric-expression}   {numeric-expression}
```

Where:

```
     image = {[+] [#]...[.][#]...[^^^^]      }
              [-]
             {alpha-variable containing image}
              the length of image <= 254
```

Discussion:

The PACK statement stores a list of numeric values in an alpha-variable in a packed decimal format specified by the image. The numeric values are packed in sequential order into the receiving alpha-variable. If the numeric values to be packed require more storage space than is provided by the alpha-variable which is to receive them, an error results.

The image determines the method by which the numeric information is stored. It provides a mechanism for retaining the original sign of the number as well as the decimal position. The image controls the storage of numeric values as follows:

1. Data is left-justified within the alpha-variable, and the decimal point is not stored during the PACK operation.

2. For each "#" character specified in the image, one digit is stored. Each individual byte of the alpha-variable contains two packed digits.

3. Fractions are truncated or extended with zeros according to the image specification. The integer portion is extended with leading zeros, if required. If the integer portion cannot be contained by the image, an error results.

PACK (cont.)

4. If a sign is specified in the image, sign information is placed in the high-order half-byte of the first byte of the receiving variable. This may contain the following values:

   0 if number is positive and exponent is positive
   1 if number is negative and exponent is positive
   8 if number is positive and exponent is negative
   9 if number is negative and exponent is negative

If no sign is specified, actual data is stored in this location.

5. The exponent, if present, is stored following the number as two hex-digits. In order to accurately represent a number, numbers in exponent format should include a sign.

6. The same image should be used to UNPACK data. Otherwise, discrepancies could occur.

7. Each packed number uses a whole number of bytes. If an image specification is used which requires only one half of the last byte, the remaining half-digit is not affected.

The PACK statement provides a method of storing large groups of numeric data to disk as alpha-variables or alpha-arrays which require less storage space if the full 14-digit precision of numbers is not a requirement for the data.

Examples:

```
0010 PACK(######) B$ FROM A8
0010 PACK(-########.####) A$() FROM B()
0010 PACK(####) STR(C$,1,6) FROM A,B,C(3)+10
0010 A$="+###.##^^^^": PACK(A$) B$ FROM A8

:0010 DIM A(2)
:0020 PACK(+####.##) A$ FROM 12.354,-123.45,1234.56
:0030 HEXPRINT A$
:0040 UNPACK(+####.##) A$ TO A,A()
:0050 PRINT A,A(1),A(2)
:RUN

0001235010123450012345602020202020

12.35          -123.45          1234.56
```

Compatibility Issues:

References:

UNPACK

$PACK

General Form:

```
     $PACK [({F} = pack-specification] buffer-variable FROM
             {D}

                                             input-value [,input-value]...
```

Where:

```
     pack-specification = {alpha-variable}
                          {literal-string}


     buffer-variable    = alpha-variable


     input-value        = {literal-string        }
                          {alpha-variable        }
                          {alpha array-variable  }
                          {numeric-expression    }
                          {numeric array-variable}
```

Discussion:

The $PACK statement is used to pack a buffer-variable with any number of values in various formats.

There are three forms of the $PACK statement:

- Delimiter Form (D parameter) each value to be packed is separated by a delimiter character in the buffer-variable.
- Field Form (F parameter) each value occupies a specified number of bytes in the buffer-variable.

$PACK (cont.)

- Internal Form (neither F nor D is specified) data is stored in standard logical record format in the buffer-variable.

Delimiter Form

In the Delimiter form, the pack-specification contains two bytes. The first byte is a control byte that must be set to a value from HEX(00) to HEX(03). This value is not used by the $PACK statement but is used by the $UNPACK statement. Refer to the $UNPACK statement for details. The second byte is a delimiter character. As the packing occurs, this delimiter character is placed between each value.

When the delimiter form is used, data values may be either alpha or numeric. Alpha data values are stored using their full dimensions or referenced length (including trailing spaces).

Numeric data values are stored in ASCII free format. Refer below (Field Form) for details on the internal structure of ASCII free format.

Examples:

```
0010 $PACK(D=HEX(032C)) Q$() FROM V$()
0010 B$=HEX(00FF)<
    :$PACK(D=B$)STR(A$(),40) FROM A$,B$,F$()

:0010 DIM X$32,A$5,B$5
:0020 A$="ABC"<
: A=1.03<
: B$="DEFG"<
: B=-3.2
:0030 $PACK(D=HEX(002C)) X$ FROM A$,A,B$,B
:0040 PRINT X$

:RUN

ABC  , 1.03,DEFG ,-3.2
```

Field Form

In the Field form, the pack-specification contains a series of two-byte format specifications for packing each value into the buffer. The first byte contains the format type. The second byte contains the length of the field.

0	0	x	x

$PACK (cont.)

The following types are allowed:

00xx	Skip xx bytes in buffer-variable
10xx	ASCII free format
2dxx	ASCII integer format
3dxx	IBM display format
4dxx	IBM USASCII - 8 format
5dxx	IBM packed decimal format
6dxx	Unsigned, packed decimal format
Axxx	Alpha field (length = xxx bytes)
Bd0x	Unsigned Binary format
Cd0x	Signed Binary format
Dd0x	Unsigned small endian
Ed0x	Signed small endian
Ft0x	Floating Point format

Where: x, xx or xxx = field width in binary (x, xx, or xxx 0)
d = implied decimal position in binary
t = class of floating point format (refer below)

An individual field specification must be specified for each input-value in the list. One field specification is specified for an array, with each element in the array being packed using that specification.

Alpha fields (Axxx) are treated as a character string with the length specified by xxx, allowing a field size up to 4075 bytes (4K-1).

The internal format of numeric fields is different for each of the numeric field specifications. In all cases, the length of the field is specified by byte two of the specification and the location of the implied decimal point (except for ASCII Free Format) is specified by the second digit of the first byte.

The $PACK statement may be used to define an entire record from a list of values. In many circumstances it is more convenient to define a RECORD structure which defines the order and FIELD format of fields within the record. This allows a program assign or inspect individual fields in the record buffer using numeric or string FIELD expressions, without any need to define or extract the entire list of fields.

$PACK (cont.)

ASCII Free Format - (10xx)

This is a displayable format equivalent to that produced by PRINTing the numeric value.

-	1.23456789	E	28

Byte 1 is the sign byte. It may be an ASCII +, - or blank.

The next series of bytes store the mantissa which may contain up to 15 digits. Digits are stored in their ASCII representation. An imbedded decimal point may be present.

Following the mantissa is the optional exponent. The exponent is represented by the letter E, followed by a sign byte (+ or -), followed by one or two ASCII digits.

The remainder of the field is filled with blanks.

For example:

```
:0010 DIM X$32
:0020 $PACK(F=HEX(10101010)) X$ FROM 1.2345678901E28,-1234567890123
:0030 PRINT X$

:RUN

1.23456789+E28 -1234567890123
```

ASCII Integer Format - (2dxx)

In this displayable format, all digits are stored as the ASCII representation of a number. The sign is contained on byte 1 of the field. The location of the decimal point is specified implicitly by the "d" parameter.

-	0010065

$PACK (cont.)

For example:

```
:0010 DIM X$32,A(3)
:0020 A(1)=-100.65
    : A(2)=33.514
    : A(3)=16.4
:0030 $PACK(F=HEX(2208)) X$ FROM A()
:0040 PRINT X$

:RUN

-0010065+0003351+0001640
```

NOTE: The decimal point is implied at two positions from the end of each field.

IBM Display Format - (3dxx)

In this form, digits are stored 1 digit per byte in the format HEX(Fd) where "d" is the digit (0-9). The sign is stored in the high-order nibble of the last byte of the field and may be "C" for positive or "D" for negative.

F0	F0	F1	F2	D3

For example:

```
:0010 DIM A$5
:0020 $PACK (F=HEX(3205))A$ FROM -1.23
:0030 HEXPRINT A$
:RUN

F0F0F1F2D3
```

IBM USASCII Format - (4dxx)

In this form, digits are stored 1 digit per byte in the format HEX(5d) where d is the digit (0-9). The sign is stored in the high-order nibble of the last byte of the field and may be A for positive or B for negative.

50	50	51	52	A3

For example:

```
:0010 DIM A$5
:0020 $PACK (F=HEX(4205))A$ FROM 1.23
:0030 HEXPRINT A$
:RUN

50505152A3
```

$PACK (cont.)

IBM Packed Decimal Format - (5dxx)

In this form, digits are stored two digits per byte in the format HEX(dd) where "d" is the digit (0-9). The sign is stored in the low-order nibble of the last byte of the field and may be C for positive or D for negative.

00	00	00	12	3D

For example:

```
:0010 DIM A$5
:0020 $PACK (F=HEX(5205))A$ FROM -1.23
:0030 HEXPRINT A$
:RUN

000000123D
```

Unsigned Packed Decimal Format - (6dxx)

In this form, digits are stored 2 digits per byte in the format HEX(dd) where d is the digit (0-9). No sign is stored.

00	00	12	34	00

For example:

```
:0010 DIM A$5
:0020 $PACK (F=HEX(6405))A$ FROM 12.34
:0030 HEXPRINT A$
:RUN

0000123400
```

Unsigned Binary Format (Bd0x)

Unsigned binary numbers may be specified using the format code HEX(Bd0x) where:

B	d	0	x

- "B" indicates that the field is binary
- "d" is a hexadecimal digit from 0 to F, indicating a number of implied decimal places in the field.
- "x" is the length of the field in bytes (values 1 to 5 permitted).

$PACK (cont.)

The following table indicates the range of values which may be stored using this format. If an implied decimal of "d" places is assumed, divide all numbers by 10^d.

Field length	Minimum	Maximum
1	0	255
2	0	65535
3	0	16777215
4	0	4294967295
5	0	1099511627775

The value stored in the field for a number "Z" is the same as that returned by BIN(INT(Z*1Ed),x).

Example:

```
:LIST
1000 DIM Z(4),Z1(4),X$14
   : Y=16000
   : Z(1)=196.32
   : Z(2)=100.305
   : Z(3)=1200
   : Z(4)=0
   : $PACK(F=HEX(B002B203)) X$ FROM Y,Z()
   : HEXPRINT X$
   : $UNPACK(F=HEX(B002B203)) X$ TO Y1,Z1()
   : PRINT Y1,Z1(1),Z1(2),Z1(3),Z1(4)
:RUN
3E80004CB000272E01D4C0000000R16000         196.32       100.3    1200
0
```

Signed Binary Format (Cd0x)

Signed binary numbers may be specified using the format code HEX(Cd0x) where:

C0	00	02	03

- "C" indicates that the field is signed binary.
- "d" is a hexadecimal digit from 0 to F indicating a number of implied decimal places in the field.
- "x" is the length of the field in bytes (values 1 to 6 permitted).

$PACK (cont.)

The following table indicates the range of values which may be stored using this format. If an implied decimal of "d" places is assumed, divide all numbers by 10^d.

Field length	Minimum	Maximum
1	-128	127
2	-32768	32767
3	-8388608	8388607
4	-2147483648	2147483647
5	-549755813888	549755813887
6	-140737488355328	140737488355327

The value stored in the field for a number "Z" is the same as that returned by BIN(INT(Z*1Ed),-x).

NOTE: Due to the way the INT() function operates on negative numbers, excess digits after those specified by the implied decimal are not truncated for negative values as would be the case for other numeric $PACK field specifications.

This difference is illustrated by the values 100.304 and -100.304 in the following example.

Example:

```
:LIST
1010 DIM Z(4),Z1(4),X$14
   : Y=16000
   : Z(1)=196.32
   : Z(2)=-100.304
   : Z(3)=100.304
   : Z(4)=1200
   :$PACK(F=HEX(C002C203)) X$ FROM Y,Z()
   :HEXPRINT X$
   :$UNPACK(F=HEX(C002C203)) X$ TO Y1,Z1()
   :PRINT Y1,Z1(1),Z1(2),Z1(3),Z1(4)
:RUN
3E80004CB0FFD8D100272E01D4C0R16000          196.32      -100.31
100.3    1200
```

$PACK (cont.)

Unsigned Small-Endian (Dd0x)

The format Dd0x is used for unsigned small-endian format, where d denotes the number of implied decimal positions and x denotes the number of bytes to be used. The number of bytes to be used may range from 1 to 5.

For example:

```
10 $PACK (F=HEX(D202)) X$ FROM 1.23
20 $UNPACK (F=HEX(D202)) X$ TO A
```

Signed Small-Endian (Ed0x)

The format Ed0x is used for signed small-endian format, where d denotes the number of implied decimal positions and x denotes the number of bytes to be used. The number of bytes to be used may range from 1 to 6.

For example:

```
10 $PACK (F=HEX(E202)) X4 FROM 1.23
20 $UNPACK (F=HEX(E202)) X$ to A
```

NOTE: "Small-endian" format is equivalent to Intel integer format.

Floating Point Format - (Ft0x)

As of Revision 3.0 of NPL, field specifications of the form HEX(Ft0x) indicate the use of a floating point field "x" bytes in length. The "t" is used to distinguish between 5 classes of floating point formats.

F	2	0	4

NOTE: This differs from other previously supported numeric field format specifications, which only supported fixed point data formats and used this hex digit to indicate an implied decimal position. In addition, only a few values of the field length "x" are supported, unlike previously supported numeric field format specifications, which allow any non-zero value for the field length. The following is a summary of the supported floating point formats:

$PACK (cont.)

"t"	Format	Acceptable values for "x"
0	Wang Internal Numeric Format	8
1	NPL Internal Numeric Format	8
2	IEEE Binary Real	H-L format
3	IEEE Binary Real	L-H format
4	DEC VAX floating point format	4, 8
5	Sortable MAT MOVE format	2,,8

In general, where both 4 and 8 are supported as a field length, the 4-byte format corresponds to a single-precision value, and the 8-byte format to a double-precision value.

NOTE: Each individual class ("t") is discussed in greater detail in following sections.

Purpose of the New Format

These formats have been implemented to allow NPL applications to read or generate numeric data in a format compatible with programs written in other languages, such as C.

Application Notes

Different computer systems use slightly different methods of representing floating point numbers internally. In addition, NPL uses its own internal format. For this reason, it is necessary to permit several different types of floating point formats.

Due to the differences between the formats, it is not always possible to store a number with complete accuracy. In such cases, the number is rounded to the nearest possible value that can be stored. If exponent underflow occurs, i.e., the number is too close to zero to be stored, the number is rounded off to zero.

Like other $PACK format codes, attempting to store numbers which are out of range for the field produces an error X71--Value exceeds format.

The following table indicates the approximate ranges of positive values that may be stored using these formats. Negative numbers have the same ranges as positive numbers, differing only in sign.

$PACK (cont.)

Format Type	Significant Digits	Minimum Denormalized	Minimum Normalized	Maximum Value
F008	13	---	1.0 E-99	9.9 E+99
F108	14	1.0 E-32768	---	1.4 E+32781
F204	7	1.4 E-45	1.2 E-38	3.4 E+38
F208	15	4.9 E-324	2.2 E-308	1.8 E+308
F304	7	1.4 E-45	1.2 E-38	3.4 E+38
F308	15	4.9 E-324	2.2 E-308	1.8 E+308
F404	7	---	2.9 E-39	1.7 E+38
F408	16	---	2.9 E-39	1.7 E+38

Note on Precision

Due to the differences between the floating point formats, there may be some loss of precision when $PACKing and $UNPACKing these formats. In particular, if a value is $PACKed into a string using any floating point binary format, $UNPACKing the string may not produce the original value. This is more likely to be noticeable when using the single-precision (4-byte) formats or when dealing with extremely small (denormalized) values.

Examples:

```
:10 DIM A$4
:20 X=123.45
:30 $PACK(F=HEX(F204)) A$ FROM X :REM note single precision floating
:40 $UNPACK(F=HEX(F204)) A$ TO Y
:50 PRINT "X = ";X
:60 PRINT "A$ = ";: HEXPRINT A$
:70 PRINT "Y = ";Y :REM stored result only accurate to about 7 digits
:RUN
X =  123.45
A$ = 42F6E666
Y =  123.449996948242

:10 DIM X$8,Y$8
:20 A=1E-322                        : REM note denormalized value
:30 $PACK(F=HEX(F208)) Y$ FROM A     : REM precision is less than
:40 $UNPACK(F=HEX(F208)) Y$ TO B     : REM normal values (15) digits
:50 PRINT "A = ";A
:60 PRINT "Y$ = ";    : HEXPRINT Y$
:70 PRINT "B = ";B
:RUN
A =  1.0000000E-322
Y$ = 0000000000000014
B =  9.8813129E-323
```

$PACK (cont.)

When dealing with floating point forms with precision exceeding that of the internal form of the RTP (14), loss of precision occurs on a $UNPACK instruction. In general, the $UNPACK routines rounds the correct number to the nearest RTP internal form value. When dealing with values very close to the maximum limits of the floating point formats, this loss of precision may cause $UNPACK to return a number which has been rounded up to a number that is actually beyond the maximum floating point limit. An attempt to $PACK the same value would produce an error X71 (value exceeds format) in this case.

Wang Internal Numeric Format - (F008)

The Wang Internal Numeric Format uses the same format used by the DATASAVE DC statement and the Internal Form of $PACK/$UNPACK. It can store decimal numbers up to 13 digits long with no errors. The size specification must be 8. The numeric value is stored as:

HEX	s	a	b	c	d	d	d	d	d	d	d	d	d	d	d	d

Where:

s = signs:
- 0 if mantissa +, exponent +
- 1 if mantissa -, exponent +
- 8 if mantissa +, exponent -
- 9 if mantissa -, exponent -

E = +/- ba = exponent (2 digits)

M = +/- c.dddddddddddd = mantissa (13 digits)

The value represented is:

$M * 10^E$

All digits must be 0-9. Numbers are always normalized so that "c" is not zero. The number zero is stored as:

HEX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$PACK (cont.)

Example:

```
:0010 DIM A$(4)8
:0020 $PACK(F=HEX(F008)) A$(1) FROM 178.125
:0030 $PACK(F=HEX(F008)) A$(2) FROM -178.125
:0040 $PACK(F=HEX(F008)) A$(3) FROM 567E-11
:0050 $PACK(F=HEX(F008)) A$(4) FROM -0.000567
:0060 LIST DIM * A$(
:RUN

(1)     "..xiP..."              HEX(0201 7812 5000 0000)
(2)     "..xiP..."              HEX(1201 7812 5000 0000)
(3)     "'.g....."              HEX(8905 6700 0000 0000)
(4)     "u.g....."              HEX(9405 6700 0000 0000)
```

NPL Internal Numeric Format - (F108)

NPL uses an internal numeric format with the advantages of Wang Internal Numeric Format, with the added benefit of faster computation times when operating on integers. Numbers are stored in the form:

HEX	m	m	m	m	m	m	m	m	m	m	m	m	e	e	e	e

Where:

M = VAL(HEX(mmmmmmmmmmmm),-6) = mantissa (12 hex digits)

E = VAL(HEX(eeee),-2) = exponent (4 hexadecimal digits)

The value represented is:

$M * 10^E$

The number is not normalized, so the same value may be represented in several ways. For details, refer to the VAL function.

$PACK (cont.)

Example:

```
:0010 DIM A$(4)8,B$8,C$8
:0020 $PACK(F=HEX(F108)) A$(1) FROM 178E42
:0030 $PACK(F=HEX(F108)) A$(2) FROM -178E42
:0040 $PACK(F=HEX(F108)) A$(3) FROM 567E-11
:0050 $PACK(F=HEX(F108)) A$(4) FROM -0.000567
:0060 LIST DIM * A$(
:0070  PRINT
:0080 PRINT "Mantissa","Exponent","Value"
:0090 B$=HEX(000000003039FFFD)
:0100 C$=HEX(00000012D644FFFB)
:0110 $UNPACK(F=HEX(F108)) B$ TO B
:0120 M=VAL(STR(B$,1,6),-6)
:0130 E=VAL(STR(B$,7,2),-2)
:0140 PRINT M,E,B
:0150 $UNPACK(F=HEX(F108)) C$ TO B
:0160 M=VAL(STR(B$,1,6),-6)
:0170 E=VAL(STR(B$,7,2),-2)
:0180 PRINT M,E,B
:RUN
(1)         "..xiP..."          HEX(0000 0002 B7CD 002A)
(2)         "..xiP..."          HEX(FFFF FFFD 4833 002A)
(3)         "'.g....."          HEX(0000 0000 0237 FFF5)
(4)         "u.g....."          HEX(FFFF FFFF FDC9 FFFA)
Mantissa        Exponent        Value
12345          -3               12.345
1234500        -5               12.345
```

IEEE Binary Real Formats - (F20x and F30x)

The IEEE P754 Standard specifies the formats of binary floating point numbers as a series of binary bits, but does not specify in what order the bytes containing those bits are to be stored in memory. Hence, computer systems may store the bytes in either H-L (High-Low) order, or in L-H (Low-High) order. The $PACK and $UNPACK statements permit either order to be specified, regardless of the computer system.

The size specification for IEEE floating point formats may be either 4 (single precision) or 8 (double precision). Double precision is capable of storing larger numbers, with greater accuracy, than single-precision numbers.

The IEEE standard defines Binary Floating point numbers as a series of binary bits in the following formats (the storage of these bits within the bytes of a string is described later):

Single precision:	S	E7	...	E2	E1	E0	F0	F1	F2	...	F22
	bit 31										bit 0
Double precision:	S	E10	...	E2	E1	E0	F0	F1	F2	...	F51
	bit 63										bit 0

Where:

$PACK (cont.)

S = sign:
 0 if positive,
 1 if negative

Ei ... E2 E1 E0 = biased exponent (binary)

F0 F1 F2 ... Fj = fraction part of mantissa (binary)

This represents the value:

$(-1)^{S} * 2^{E-B} * (1 . F0\ F1\ F2 ... Fj)$

Where:

S = sign
E = biased exponent = Ei ... E2 E1 E0 (binary)
B = exponent bias:
 127 for single precision reals,
 1023 for double precision reals

1 . F0 F1 F2 ... Fj = mantissa (binary)

This is called the normalized form.

NOTE: The "binary point" is placed immediately to the right of the leading "1". The leading "1" is implicit and is not stored.

There are four types of values which cannot be stored as normalized numbers. Their signs may be either positive or negative:

Zero:	E = 000... ,	F = 000...
Denormalized:	E = 000... ,	F = any non-zero bit pattern
NaN:	E = 111... ,	F = any non-zero bit pattern
Infinity:	E = 111... ,	F = 000...

$PACK (cont.)

Where:

F = F0 F1 F2 ... Fj (binary)

E = Ei ... E2 E1 E0 (binary)

"non-zero bit pattern" means at least one bit must be a "1".

NPL does not distinguish between positive and negative zero. $PACK stores zero as a "positive" zero; $UNPACK ignores the sign of zero values.

"Denormalized" numbers represent values very close to zero. They have an implied leading bit of "0", and represent the value:

$(-1)^S * 2^K * (0 . F0\ F1\ F2... Fj)$

Where:

S = sign
K = minimum exponent:
-126 for single-precision reals,
-1022 for double-precision reals

"Infinity" may be generated by a coprocessor as the result of operations such as division by zero.

"NaN" (Not-a-Number) may be generated by a coprocessor as the result of meaningless operations such as infinity subtracted from infinity.

NPL does not have internal numeric values equivalent to Infinity or NaN, so $PACK does not generate those values. An attempt to $UNPACK a NaN or Infinity produces an error X75-Illegal number.

$PACK (cont.)

IEEE Binary Real H-L Format, Single Precision - (F204)

IEEE Binary Real Single Precision H-L Format is normally used on 68000-series computers and floating point coprocessors such as the MC68881. The number is stored in 4 bytes in the following format:

bit 7								bit 0
1st byte	S	E7	E6	E5	E4	E3	E2	E1
2nd byte	E0	F0	F1	F2	F3	F4	F5	F6
3rd byte	F7	F8	F9	F10	F11	F12	F13	F14
4th byte	F15	F16	F17	F18	F19	F20	F21	F22

Examples:

```
:0010 DIM A$(4)4
:0020 $PACK(F=HEX(F204)) A$(1) FROM 178.125
:0030 $PACK(F=HEX(F204)) A$(2) FROM -178.125
:0040 $PACK(F=HEX(F204)) A$(3) FROM (250020/2048)
:0050 $PACK(F=HEX(F204)) A$(4) FROM (-7/1024)
:0060 LIST DIM * A$(
:RUN

(1)     "C2 ."          HEX(4332 2000)
(2)     "C2 ."          HEX(C332 2000)
(3)     "Bt)."          HEX(42F4 2900)
(4)     ";'.."          HEX(BBE0 0000)
```

$PACKed strings in this format may be compared directly without $UNPACKing them. If both values are positive, the result of the comparison correctly reflects the relative values of the $PACKed numbers. If one or both of the values is negative, the meaning of ">" and "<" is reversed.

IEEE Binary Real L-H Format, Single Precision - (F304)

IEEE Binary Real Single Precision L-H Format is normally used on 80286/80386 based computers and floating point coprocessors such as the 80287/80387 and the NS32081. The number is stored in 4 bytes in the following format:

bit 7								bit 0
1st byte	F15	F16	F17	F18	F19	F20	F21	F22
2nd byte	F7	F8	F9	F10	F11	F12	F13	F14
3rd byte	E0	F0	F1	F2	F3	F4	F5	F6
4th byte	S	E7	E6	E5	E4	E3	E2	E1

$PACK (cont.)

NOTE: This is similar to the (F204) format, except that the order of the bytes is reversed. Because the order is reversed, string comparisons have no meaningful results.

Examples:

```
:0010 DIM A$(4)4
:0020 $PACK(F=HEX(F304)) A$(1) FROM 178.125
:0030 $PACK(F=HEX(F304)) A$(2) FROM -178.125
:0040 $PACK(F=HEX(F304)) A$(3) FROM (250020/2048)
:0050 $PACK(F=HEX(F304)) A$(4) FROM (-7/1024)
:0060 LIST DIM * A$(
:RUN

(1)     ". 2C"          HEX(0020 3243)
(2)     ". 2C"          HEX(0020 32C3)
(3)     ".)tB"          HEX(0029 F442)
(4)     "..';"          HEX(0000 E0BB)
```

IEEE Binary Real H-L Format, Double Precision - (F208)

IEEE Binary Real Double Precision H-L Format is normally used on 68000-series computers and floating point coprocessors such as the MC68881. The number is stored in 8 bytes in the following format:

bit 7								bit 0
1st byte	S	E10	E9	E8	E7	E6	E5	E4
2nd byte	E3	E2	E1	E0	F0	F1	F2	F3
3rd byte	F4	F5	F6	F7	F8	F9	F10	F11
4th byte	F12	F13	F14	F15	F16	F17	F18	F19
5th byte	F20	F21	F22	F23	F24	F25	F26	F27
6th byte	F28	F29	F30	F31	F32	F33	F34	F35
7th byte	F36	F37	F38	F39	F40	F41	F42	F43
8th byte	F44	F45	F46	F47	F48	F49	F50	F51

Examples:

```
:0010 DIM A$(4)8
:0020 $PACK(F=HEX(F208)) A$(1) FROM 178.125
:0030 $PACK(F=HEX(F208)) A$(2) FROM -178.125
:0040 $PACK(F=HEX(F208)) A$(3) FROM (250020/2048)
:0050 $PACK(F=HEX(F208)) A$(4) FROM (-7/1024)
:0060 LIST DIM * A$(

:RUN

(1)     "@fD....."              HEX(4066 4400 0000 0000)
(2)     "@fD....."              HEX(C066 4400 0000 0000)
(3)     "@^......"              HEX(405E 8520 0000 0000)
(4)     "?......."              HEX(BF7C 0000 0000 0000)
```

$PACK (cont.)

$PACKed strings in this format may be compared directly without $UNPACKing them. If both values are positive, the result of the comparison correctly reflects the relative values of the $PACKed numbers. If one or both of the values is negative, the meaning of ">" and "<" is reversed.

IEEE Binary Real L-H Format, Double Precision - (F308)

IEEE Binary Real Double Precision L-H Format is normally used on 80286/80386 based computers and floating point coprocessors such as the 80287/80387 and the NS32081. The number is stored in 4 bytes in the following format:

bit 7								bit 0
1st byte	F44	F45	F46	F47	F48	F49	F50	F51
2nd byte	F36	F37	F38	F39	F40	F41	F42	F43
3rd byte	F28	F29	F30	F31	F32	F33	F34	F35
4th byte	F20	F21	F22	F23	F24	F25	F26	F27
5th byte	F12	F13	F14	F15	F16	F17	F18	F19
6th byte	F4	F5	F6	F7	F8	F9	F10	F11
7th byte	E3	E2	E1	E0	F0	F1	F2	F3
8th byte	S	E10	E9	E8	E7	E6	E5	E4

NOTE: This is similar to the (F208) format, except that the order of the bytes is reversed. Because the order is reversed, string comparisons have no meaningful results.

Examples:

```
:0010 DIM A$(4)8
:0020 $PACK(F=HEX(F308)) A$(1) FROM 178.125
:0030 $PACK(F=HEX(F308)) A$(2) FROM -178.125
:0040 $PACK(F=HEX(F308)) A$(3) FROM (250020/2048)
:0050 $PACK(F=HEX(F308)) A$(4) FROM (-7/1024)
:0060 LIST DIM * A$(
  :RUN

(1)     ".....Df@"             HEX(0000 0000 0044 6640)
(2)     ".....Df@"             HEX(0000 0000 0044 66C0)
(3)     "......^@"             HEX(0000 0000 2085 5E40)
(4)     ".......?"             HEX(0000 0000 0000 7CBF)
```

$PACK (cont.)

DEC VAX Floating Point Format - (F40x)

The size specification for DEC VAX floating point formats may be either 4 (F_floating) or 8 (D_floating). D_floating is capable of storing numbers with greater accuracy than F_floating numbers. Floating point numbers on the DEC VAX are defined as a series of binary bits in the following formats (the storage of these bits within the bytes of a string is described later):

F_float:	S	E7	...	E2	E1	E0	F0	F1	F2	...	F22
	bit 31										bit 0
D_float:	S	E7	...	E2	E1	E0	F0	F1	F2	...	F54
	bit 63										bit 0

This represents the value:

$(-1)^S * 2^{E-128} * (0 . 1 F0 F1 F2 ... Fj)$

Where:

S = sign:
0 if positive,
1 if negative

E = biased exponent = Ei ... E2 E1 E0 (binary)

0 . 1 F0 F1 F2 ... Fj = mantissa (binary)

NOTE: The "binary point" is immediately to the left of the first "1" bit. The leading "0.1" is implied and is not stored.

A floating point number with a biased exponent of 0 is a special case. If the sign bit is 0, it represents the number zero, regardless of what is in the mantissa. If the sign bit is 1, it is an invalid number, and attempting to $UNPACK it produces an error X75-Illegal Number.

$PACK (cont.)

DEC VAX F_floating Format - (F404)

An F_floating number is stored in 4 bytes in the following format:

	bit 7							bit 0
1st byte	E0	F0	F1	F2	F3	F4	F5	F6
2nd byte	S	E7	E6	E5	E4	E3	E2	E1
3rd byte	F15	F16	F17	F18	F19	F20	F21	F22
4th byte	F7	F8	F9	F10	F11	F12	F13	F14

Examples:

```
:0010 DIM A$(4)4
:0020 $PACK(F=HEX(F404)) A$(1) FROM 178.125
:0030 $PACK(F=HEX(F404)) A$(2) FROM -178.125
:0040 $PACK(F=HEX(F404)) A$(3) FROM (250020/2048)
:0050 $PACK(F=HEX(F404)) A$(4) FROM (-7/1024)
:0060 LIST DIM * A$(
:RUN

(1)     "2D. "          HEX(3244 0020)
(2)     "2D. "          HEX(32C4 0020)
(3)     "tC.)"          HEX(F443 0029)
(4)     "‘.<."          HEX(E0BC 0000)
```

DEC VAX D_floating Format - (F408)

An D_floating number is stored in 8 bytes in the following format:

	bit 7							
1st byte	E0	F0	F1	F2	F3	F4	F5	F6
2nd byte	S	E7	E6	E5	E4	E3	E2	E1
3rd byte	F15	F16	F17	F18	F19	F20	F21	F22
4th byte	F7	F8	F9	F10	F11	F12	F13	F14
5th byte	F31	F32	F33	F34	F35	F36	F37	F38
6th byte	F23	F24	F25	F26	F27	F28	F29	F30
7th byte	F47	F48	F49	F50	F51	F52	F53	F54
8th byte	F39	F40	F41	F42	F43	F44	F45	F46

$PACK (cont.)

Examples:

```
:0010 DIM A$(4)8
:0020 $PACK(F=HEX(F408)) A$(1) FROM 178.125
:0030 $PACK(F=HEX(F408)) A$(2) FROM -178.125
:0040 $PACK(F=HEX(F408)) A$(3) FROM (250020/2048)
:0050 $PACK(F=HEX(F408)) A$(4) FROM (-7/1024)
:0060 LIST DIM * A$(
:RUN

(1)     "2D. ...."             HEX(3244 0020 0000 0000)
(2)     "2D. ...."             HEX(32C4 0020 0000 0000)
(3)     "tC.)...."             HEX(F443 0029 0000 0000)
(4)     "'<......"             HEX(E0BC 0000 0000 0000)
```

Examples:

```
    0010 Q$=HEX(A004A0105209): $PACK(F=Q$) Q1$() FROM A$,A0$,A
    0010 $PACK(F=HEX(5001)) T1$() FROM A2
    0010 $PACK(F=HEX(0001A00500085204)) Q1$() FROM A0$,A0

    :0010 DIM X$32,Y$32,Y(5),A$3,B$(3)5
    :0020 $PACK(F=HEX(52055205520552055205)) X$ FROM -123.45,123.456,99,-99,34
    :0030 LIST DIM *X$
    :0040 Y$=ALL(FF)
    :0050 $PACK(F=HEX(A00300045205A005A005A005A005)) Y$ FROM
    "abc",123.45,"defg","hijk","lmno"
    :0050 LIST DIM *Y$

    :RUN

DIM X$32
       "..?4]..?4\......"  HEX(0000 1234 5D00 0012 345C 0000 0990 0C00)

STR(17)".......@.      "  HEX(0009 900D 0000 0340 0C20 2020 2020 2020)

DIM Y$32
       "abc    ..?4\defg"  HEX(6162 63FF FFFF FF00 0012 345C 6465 6667)
STR(17)" hijk lmno      "  HEX(2068 696A 6B20 6C6D 6E6F 20FF FFFF FFFF)
```

$PACK (cont.)

Field Format (F50x)

In this format, x represents the number of bytes in the alpha variable to be used.

The alpha representation of numeric values by this format is identical to that produced by MAT MOVE. Refer to the MAT MOVE discussion for details. The advantage of this format is that it produces alphanumeric values that can be sorted even if the numeric values contain both positive and negative values.

For example:

```
    10 DIM A$8
    20 $PACK(F=HEX(F508)) A$ FROM -1.234
    30 $UNPACK(F=HEX(F508)) A$ to A
```

The size of the alpha variable used to store the result affects the precision of the operation. A size of eight bytes ensures full precision (13 digits). For smaller values, the precision can be calculated by the formula D= n^{*2-3} where n is the number of bytes in the alpha variable and D is the number of digits of precision.

The Internal Form

The Internal form of $PACK uses the same format used by the DATASAVE DC statement when saving a record to disk. The $PACK operation is terminated when all values in the list have been sequentially packed. Refer to Section 7.3.7 of the Programmer's Guide for further details on the internal logical disk record format.

```
    0010 $PACK A$() FROM B$,C(),D$(3),E
       0010 $PACK X$ FROM Z(2),Z(3),Z$,Z(4)

       :0010 DIM A$(1)20
       :0020 $PACK A$() FROM 12.345, "ABCD"
       :0030 LIST DIM *A$()
       :RUN

    DIM A$(1)20
    (1)    ".....#E....ABCD"   HEX(8001 0801 0123 4500 0000 0084 4142 4344)
    STR(17)". "                HEX(FD20 2020)
```

NOTE: Refer to Chapter 4, Library Functions, for further information about $PACK.

Compatibility Issues:

The unsigned binary (Bd0x) and signed binary (Cd0x) field format specifications are supported only on NPL Revision 2.1 or greater and are not supported on the Wang 2200.

The floating point (Ft0x) field format specification is supported only on NPL Revision 3.0 or greater and is not supported on the Wang 2200.

$PACK (cont.)

The extended alpha field format (Axxx) is supported in Release IV of NPL or greater. Previous releases of NPL support an alpha format of (A0xx).

Floating point format F50x is supported only by NPL revision 3.01.11 or later.

Little-endian formats (Dxxx and Cxxx) are supported only by NPL Revision 3.01.13 or later.

References:

DATA SAVE DC
FIELD
Numeric FIELD Assignment
Numeric FIELD Expression
String FIELD Assignment
String FIELD Expression
MAT MOVE
RECORD
$UNPACK

#PART Function

General Form:

```
#PART
```

Discussion:

The #PART function returns the partition number of the current user partition. #PART is typically used to distinguish between users in a multi-user environment. This is valid wherever a numeric-expression is legal.

Examples:

```
0010 A=B+#PART
0010 C1(1)=#PART
```

Compatibility Issues:

The generation of #PART is hardware-specific. Refer to appropriate NPL Supplement for details on the hardware system.

References:

#PI Function

General Form:

```
#PI
```

Discussion:

The #PI function returns the mathematical constant "pi", which has a value 3.1415926535898. This is valid wherever a numeric-expression is legal.

Examples:

```
0010 M5 = #PI*R4*2
0010 A = 90/#PI
```

Compatibility Issues:

The value of #PI on the Wang 2200 is 3.14159265359.

The value of #PI may be modified by byte 34 of the $OPTIONS system variable. Refer to $OPTIONS for details.

References:

$OPTIONS

POS Function

General Form:

```
     POS ([-] search-value rel-op character-value)
```

Where:

```
     search-value    = {alpha-variable}
                       {literal-string}

     character-value = {alpha-variable}
                       {literal-string}
                       {hh            }

     rel-op          = relational operator { <,=,>,<=,>=,<> }

     h               = hexadecimal digit (0-9 or A-F)
```

Discussion:

The POS function returns the position of the first occurrence of a character value in the search-value that satisfies the specified relationship. This is valid wherever a numeric expression is legal.

A search can be made for the first character equal to, less than, greater than, less than or equal to, greater than or equal to, or not equal to a character. The character used is the first character of the character-value, or HEX(hh) if character-value is expressed as two hex-digits.

The "-" parameter is used to return the position of the last occurrence of a character-value in the search-value that satisfies the specified relationship. For either search direction, if a character that satisfies the specified relationship cannot be found, a value of zero is returned.

The POS function is a numeric function (returns a numeric value) which can be used wherever a numeric-expression is legal.

POS Function (cont.)

Examples:

```
0010 X = POS(X$<>"A")
0010 W = POS(STR(A$,3)>Q$)
0010 Q=POS("12340"=C$)
0010 IF POS("ABCDEF"=W$)=0 THEN 1000
0010 R = POS(STR(W$(4),15,4)=STR(P9$,,4))
0010 A = POS('employee$="Smith")
0010 A = POS("ABCDE" = 'whatchar$)
0010 A = POS (employee.record$.Name$ = "Green")
:0010 A$="Niakwa NPL"
    : B$="a"
    : PRINT POS (A$=B$)
    : PRINT POS (-A$=B$)
    : PRINT POS (STR(A$,8)<"A")
    : PRINT POS ("ABCDEabcde">=B$)

:RUN
 3
 9
 6
 6
```

Compatibility Issues:

References:

PRINT

General Form:

```
     PRINT [item] [{,} [item]]...
                   {;}
```

Where:

```
     item = {AT Function       }
            {BOX Function      }
            {HEXOF Function    }
            {TAB Function      }
            {numeric-expression}
            {alpha-variable    }
            {literal-string    }
```

Discussion:

The PRINT statement is used to print the specified item(s) to the currently selected PRINT device.

During program execution, PRINT output is sent to the current default PRINT address in the Internal Device Table. The line width value assigned to the default PRINT address is used. From Immediate Mode, PRINT output is sent to the screen (as a convenience for debugging). Line width for output from Immediate Mode PRINT commands is always 80 columns.

The comma and semicolon delimiters may be used to control print output. A comma forces a tab to the next "column", where columns are started every 16 characters on the output line. A semicolon suppresses tabbing and places consecutive items immediately after one another. At the end of a print statement, either delimiter suppresses the start of a new line; therefore, PRINT output from subsequent statements would follow that of the current PRINT statement on the same output line.

Numeric values are always printed or displayed in one of two ways:

1. Normal Numeric Format

2. Scientific Notation Format

PRINT (cont.)

Normal Numeric Format is used for numeric values whose absolute value is greater than .1 and is 1 to 15 digits in length (plus the decimal point), or which can be represented by a non-exponential format with 15 digits or less (e.g., .001). Numeric values printed in normal numeric format occupy a maximum of 18 character spaces:

1 character	a leading space for either a minus sign or blank
up to 15 characters	15 digits
up to 1 character	a decimal point
1 character	the trailing space
up to 18 characters maximum	

Numeric values printed in normal numeric format occupy a minimum of 3 character spaces (the leading space, the digit, and the trailing blank).

When a numeric value has an absolute value less than .1, and cannot be represented by a non-exponential format with 15 or fewer digits, Scientific notation format is used.

A numeric value printed in scientific notation format occupies exactly 16 character spaces: a leading space for either a minus sign for negative values or blank for positive values, one integer digit, a decimal point, 8 decimal digits, the letter "E", the sign of the exponent, a two-digit exponent, and a trailing blank.

NOTE: Only 9 significant digits can be displayed when printing in scientific notation format (e.g., 1.23456789E+12).

The rate of PRINT output when printing to the screen can be reduced by using the SELECT P statement. SELECT P allows the user to insert a pause while printing to the screen in the range of 1/6 second to 1 1/2 seconds (SELECT P1 through SELECT P9).

Generation of Carriage Returns:

A carriage return (HEX(0D)) is automatically inserted into PRINT output under the following circumstances:

1. Following the last item in the PRINT statement, unless a trailing comma or semicolon is present.

PRINT (cont.)

2. Before printing a numeric-expression, literal, or alpha-variable, if the length causes the value to "wrap" onto a new line, based on the currently defined line width.

Effect of Device Type:

The device type (first character of device address) affects print output as follows:

- Device Type "0" - A linefeed (HEX(0A)) is automatically output following each carriage return (HEX(0D)). This device type is normally used in PRINTing to the screen (/005).

- Device Type "2" - No extra character is inserted following each carriage return. This device type is normally used for printers (/215 for example).

- Device Type "4" - A linefeed (HEX(0A)) is automatically output following each carriage return (HEX(0D)). However, the automatic generation of a linefeed is suppressed at the end of PRINT statements.

- Device Type "7" - No character is appended following a carriage return.

NOTE: **The RunTime Package normally adds a line feed to each carriage return (HEX(0D)) sent to a printer type device selected as 2xx. This ensures that /2xx output is single spaced. Printer conventions on a Wang 2200 system assume that the printer does its own line feed. To overstrike a line, use a 2xx device address in conjunction with the ALF=N option in the $DEVICE specification.**

The PRINT functions AT, BOX, TAB and HEXOF are discussed in separate sections.

Examples:

```
0010 PRINT STR(Q1$(),40,10);
0010 PRINT A$,B$,C$
0010 PRINT "Niakwa NPL wants you!!!"
0010 PRINT A+B+C/D
0010 PRINT HEX(0E);TAB(10);"The Amount is ";A
0010 PRINT A,B,C$,D,
0010 PRINT "Mr. ";A$;" you have just won $1,000,000 dollars!!"

:0010 A=10
    : B=20
    : C$="Niakwa"
:0020 PRINT A,B,C$
    : PRINT A;B;C$
:RUN
10            20              Niakwa
10  20 Niakwa
```

PRINT (cont.)

Compatibility Issues:

The effect of the PRINT statement may vary on different hardware systems. On the Wang 2200, the maximum default print size is 16 positions since numbers have, at most, 13-digit precision. Refer to the appropriate NPL Supplement for details.

References:

PRINT AT function
PRINT BOX function
PRINT HEXOF function
PRINT TAB function
SELECT function
Internal Device Table - Section 7.2.3 of the Programmer's Guide

PRINT AT Function

General Form

```
PRINT AT(row,column[,[length]])
```

Where:

```
row    = a numeric-expression specifying screen row.

column = a numeric-expression specifying screen column.

length = a numeric-expression specifying number of characters to
         be erased.
```

Discussion:

The PRINT AT function is used to position the cursor at the specified row and column on the screen, optionally erasing all or part of the remaining screen. The destination position is specified by giving the row and column of the desired location. The rows and columns are numbered starting at 0. All terminals supported by NPL support at least rows 0-23 and columns 0-131. Some terminals may support additional rows or columns. Refer to Appendix D of the Programmer's Guide for information on supported terminals.

The optional length parameter specifies the number of characters to be erased on the screen starting from the destination position of the cursor. The cursor is first positioned by the first two expressions, the number of characters specified is then blanked out, and the cursor is repositioned to the start of the blanked-out area.

If the length parameter is omitted but the second comma is present, the remainder of the screen is erased beginning at the destination position of the cursor.

The size of the screen should be correctly set by the SELECT LINE parameter and the width as the current SELECT PRINT, since these are used by the system when erasing the "rest of screen".

Multiple PRINT functions, such as PRINT AT, BOX, TAB and HEXOF may be combined into one PRINT statement, separated by semicolons.

PRINT AT Function (cont.)

Examples:

```
0010 PRINT AT(10,20,30);A$
0010 PRINT C$;AT(5,A);
0010 PRINT AT(3,0,80);
0010 PRINT B$;AT(A,B,C);A$
0010 PRINT AT(8,0,);A$;B$;C$
0010 PRINT AT(12,22);A$
```

Compatibility Issues:

References:

PRINT
PRINT BOX function
PRINT HEXOF function
PRINT TAB function

PRINT BOX Function

General Form:

```
     PRINT BOX(height,width)
```

Where:

```
     height = numeric-expression which specifies the height of the box
              (in lines).

      width = numeric-expression which specifies the width of the box
              (in character positions).
```

Discussion:

The PRINT BOX function is used to draw or erase a box of the specified height and width on a screen. The PRINT BOX function uses the current cursor position as the upper-left corner of the box.

The box is drawn or erased depending upon the sign of the expressions. If both expressions are positive, the box is drawn. If both expressions are negative, the box is erased. If one expression is positive and the other expression is negative, a RunTime error is generated (P34). Specifying a height of 0 causes a horizontal line to be drawn (or erased depending on the specified width), while specifying a width of 0 causes a vertical line to be drawn (or erased depending on the sign of specified height).

Boxes which are too large to fit on the screen are suppressed from printing.

If printing a box would cause the screen to scroll, the screen is scrolled enough to print the box, assuming "true" box graphics are available.

There are two kinds of boxes which may be displayed: "true" boxes and "character" boxes.

PRINT BOX Function (cont.)

"True" boxes require a monitor which can support dual text and graphics mode. "Character" boxes are printed when the dual text and graphics mode is not available, and prints boxes with a selectable replacement character. The primary difference between "true" boxes and "character" boxes is that horizontal lines print between character rows when "true" boxes are used. However, when "character" boxes are used, horizontal lines must occupy a full character row, which means that a blank row above and below the text to be boxed must be allocated, or screen scrolling may occur when the character box is printed. Refer to the $BOXTABLE system variable for details on selecting the type of boxes to print, and the replacement character for the boxes.

Multiple PRINT functions, such as PRINT AT, BOX, TAB and HEXOF may be combined into one PRINT statement, separated by semicolons.

Examples:

```
0010 PRINT AT(5,10);BOX(5,5)
0010 PRINT BOX(-4,-8)
0010 PRINT BOX(23,0)
0010 PRINT BOX(0,12)
```

Compatibility Issues:

The implementation of box graphics is highly operating system-dependent. Refer to the NPL Supplement(s) for details on the availability of "true" boxes and character set used for "character" boxes.

In NPL, boxes which would cause the screen to scroll up are suppressed from printing. Prior to Revision 2.00, boxes which would cause the screen to scroll were suppressed. Should a PRINT BOX statement which would cause the screen to scroll be executed, it is treated as no operation. The Wang 2200 would actually scroll the screen to print the specified box.

References:

$BOXTABLE
PRINT
PRINT AT function
PRINT HEXOF function
PRINT TAB function

PRINT HEXOF Function

General Form:

```
PRINT HEXOF ({literal-string})
             {alpha-variable}
```

Discussion:

The PRINT HEXOF function is used to print the hexadecimal value of an alpha-variable or literal-string. All characters of an alpha-variable are displayed, including trailing spaces.

Multiple PRINT functions, such as PRINT AT, BOX, TAB and HEXOF may be combined into one PRINT statement, separated by semicolons.

Examples:

```
0010 PRINT HEXOF("?")
0010 PRINT HEXOF(STR(Q1$(),20,40))
0010 PRINT "disk device",HEXOF(D$)
0010 PRINT HEXOF(A$())
0010 PRINT HEXOF('anyString$(1))
0010 PRINT HEXOF(menu$.option$)
:0010 X$="Niakwa"
:0020 PRINT X$
:0030 PRINT HEXOF(X$)
:RUN
Niakwa
4E69616B77612020202020202020202020
```

Compatibility Issues:

References:

PRINT
PRINT AT function
PRINT BOX function
PRINT TAB function
HEXPRINT

PRINT SCREEN

General Form:

```
     PRINT SCREEN alpha-variable [,AT (x,y)][,BOX (r,c)]
```

Where:

```
     x = a numeric-expression specifying the starting row.

     y = a numeric-expression specifying the starting column.

     r = a numeric-expression specifying the number of rows to
         print. For any value r, r+1 rows are input.

     c = a numeric-expression specifying the number of columns per row
         to print.  For any value c, c+1 columns are printed for each
         row printed.
```

Discussion:

PRINT SCREEN is used to print the specified portion of the screen from the specified variable. The screen is printed row by row starting at the specified x,y coordinates for the specified number of rows and columns.

Information in the specified variable is assumed to be in a format compatible with that produced by INPUT SCREEN (refer to INPUT SCREEN for a detailed description of this format). PRINT SCREEN is automatically set current video parameters to those stored in the header information fields of the specified variable. In addition, AT and BOX values stored in the header information field are automatically be used unless explicit AT or BOX values are specified.

NOTE: Care must be taken if explicit AT or BOX values are specified. If AT and BOX values specified would cause the displayed area to exceed the screen size (either rows or columns), a P34 error (Illegal Value) results.

PRINT SCREEN (cont.)

PRINT SCREEN displays only complete sections (refer to INPUT SCREEN for a detailed description of the size and contents of all sections). If the specified variable is not large enough to contain all sections, partial sections are not used. If the color section is not used, background/foreground colors are not those used at the time of the most recent terminal reset. If the attributes/box graphics section is not used, the "normal" video mode is used and no boxes are displayed. If the character section is not used, all spaces are displayed.

NOTE: Calculation of section sizes is based on BOX values specified to PRINT SCREEN. Therefore, if explicit BOX values are specified and these values do not match the BOX values used when the variable containing the screen image was generated, unpredictable results occur.

PRINT SCREEN always clears the specified portion of the screen before displaying any new contents.

PRINT SCREEN is primarily intended to be used in conjunction with INPUT SCREEN to temporarily save and then redisplay a portion of the screen. This capability allows new "pop-up" type features to be added to existing applications.

For example, the following routine could be added to an existing program:

```
0010 DIM A$80+(10+1)*(20+1)*3,B$1
   : REM Dimension A$ large enough to hold a 10 by 20 area.
0020 INPUT SCREEN A$, AT(0,50),BOX(10,20)
   : REM Save existing screen portion
0030 PRINT SCREEN STR(A$,,80)
   : REM Blank out the area
0040 PRINT AT(0,50);BOX(10,20);AT(1,51);"POP-UP Area"
0050 REM Perform other work in the "pop-up" area
0090 KEYIN B$
   : REM Display "pop-up" area until a key is pressed
0100 PRINT SCREEN A$
   : REM Restore original area
```

In the example above, the following points are worth noting:

1. Variable A$ is dimensioned to exactly the size required to store all sections for the BOX values to be used (10 by 20).
2. The use of PRINT SCREEN to clear an area of the screen is shown on line 30.

PRINT SCREEN (cont.)

NOTE: The area cleared corresponds exactly to the AT and BOX values specified for INPUT SCREEN at line 20. This is because the values stored in the header fields of A$ (as established by INPUT SCREEN at line 20) are used to determine the area to clear.

3. The PRINT BOX statement at line 40 is used to draw a box around the "pop-up" area of the screen.

NOTE: In cases where the size and location of the pop-up area are not known, this information could be extracted from the header fields of the variable used in INPUT SCREEN at line 20.

Also, the values for PRINT AT and BOX on line 40 correspond exactly to the AT and BOX values used for INPUT SCREEN.

4. While displaying information in the "pop-up" area, the application may freely use video attributes and colors. Previously used attributes and colors are automatically regenerated by PRINT SCREEN at line 100 when the original screen area is restored.

Another typical use of INPUT SCREEN/PRINT SCREEN is to preserve the screen contents for redisplay after $SHELL.

For example:

```
:0010  DIM A$80+25*80*3 :REM Large enough for 24*80 screen, all 3 sections
:0020  DIM X$50
:0030  PRINT AT(12,0,80);
:0040  LINPUT "Enter command "-X$
:0050  INPUT SCREEN A$
:0060  $SHELL X$
:0070  PRINT SCREEN A$
:0080  GOTO 30
:PRINT HEX(03)
:LIST
:RUN
```

PRINT SCREEN may be used to display screen images that have been created by INPUT SCREEN and stored on disk. However, some special considerations apply to this technique:

1. If different screen translation is in effect when PRINT SCREEN is executed, the characters displayed by PRINT SCREEN may be different from the characters displayed when INPUT SCREEN was executed.

PRINT SCREEN (cont.)

2. If PRINT SCREEN is executed on a different monitor type from the one on which INPUT SCREEN was executed, the resulting screen display may be different due to terminal differences. Refer to Appendex D of the Programmer's Guide for details on terminal characteristics.

3. Future releases of the non-interpretive RunTime do not necessarily accept older level versions of INPUT SCREEN as data to PRINT SCREEN. Therefore, the application must check to make sure that the proper revision of the RunTime is in use when attempting to display stored data.

Because of the above restrictions, use of INPUT SCREEN/PRINT SCREEN as a quick method for generating complex screens is not a good technique.

HINT: It is recommended that INPUT SCREEN/PRINT SCREEN be used to temporarily store and then redisplay a portion of the screen, allowing that portion of the screen to be temporarily used for some other purpose, as illustrated in the example above.

Examples:

```
0010 PRINT SCREEN A$
0010 PRINT SCREEN A$, BOX(5,10)
0010 PRINT SCREEN A$, AT(3,20), BOX(5,10)
0010 PRINT SCREEN STR(A$,,80)
0010 PRINT SCREEN STR(A$,24,236), AT(B-A+1,C-D+1), BOX(3,12)
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

PRINT SCREEN is not supported in the Wang 2200.

References:

INPUT SCREEN
Screen Handling - Chapter 7 of the Programmer's Guide
Terminal Characteristics - Appendix D of the Programmer's Guide

PRINT TAB Function

General Form:

```
PRINT TAB(expression)
```

Where:

```
0 <= expression <= 255
```

Discussion:

The PRINT TAB function positions the cursor or column counter to the specified column position. If the specified column is larger than the selected output line width, the counter is positioned to the first column of the next line. If the specified position is less than the current output tabs, the TAB() is ignored.

On the screen, blanks are displayed up to the desired tab position. Consequently, intervening values displayed on the line are erased.

NOTE: **The system maintains an internal TAB() counter for printed output. This counter is cleared to 0 by each SELECT PRINT statement, and incremented for each character printed in the range HEX(10) to HEX(FF). It is cleared to 0 when a carriage return (HEX(0D)) is printed. Sending control sequences to a printer with codes outside the range HEX(00-0F) may cause the TAB counter to disagree with the actual print position.**

Multiple PRINT functions, such as PRINT AT, BOX, TAB and HEXOF may be combined into one PRINT statement, separated by semicolons.

Examples:

```
    0010 PRINT TAB(30);
    0010 PRINT TAB(X+Y)
    0010 PRINT X;TAB(20);Y
    0010 PRINT X$;TAB(20);HEXOF(Y$)
```

Compatibility Issues:

References:

PRINT
PRINT AT function
PRINT BOX function
PRINT HEXOF function

PRINT TO

General Form:

```
PRINT TO alpha-variable [{,} [item]]...
                         {;}
```

Where:

```
item = {AT Function       }
       {BOX Function      }
       {HEXOF Function    }
       {TAB Function      }
       {numeric-expression}
       {alpha-variable    }
       {literal-string    }
```

Discussion:

The PRINT TO statement formats a list of variables according to the referenced function, using the same logic as the PRINT statement. However, instead of printing the result, this statement stores the result in an alpha-variable for later use.

The first two bytes of the alpha-variable are used to store a count of characters (in binary) placed in the alpha-variable by the PRINT TO statement. Whenever the PRINT TO statement is executed, this count is retrieved to determine the starting location within the alpha-variable to begin storing output. The count is then updated to reflect the new count of characters in the alpha-variable. This allows multiple PRINT TO operations to place successive buffered output in the correct location. PRINT TO statements may be intermixed with PRINTUSING TO statements referencing the same receiver-variable.

The count should be initialized to HEX(0000) before using an alpha-variable for the first time or before reusing an alpha-variable.

If the total number of characters to be stored in the alpha-variable would exceed the length of the alpha-variable, output is truncated and the count is set to the length of the alpha-variable.

If the buffer count value exceeds the actual size of the specified buffer variable at the start of the statement, a P52 error (Variable or Value Too Short) is generated.

PRINT TO (cont.)

If an error occurs when evaluating arguments in the PRINT TO statement, the count value in the first two bytes of the alpha-variable may not include previous arguments in the same statement.

NOTE: No character is inserted following carriage returns, and that carriage returns are not automatically generated when line width is exceeded.

PRINT TO would typically be used to capture output which otherwise would have been printed to a print class device other than the screen.

HINT: The recommended technique for capturing output to the screen is to PRINT the output to the screen in a normal fashion and then use INPUT SCREEN to read the screen contents into a variable.

NOTE: Use of AT and BOX functions (which are intended for screen output) is supported for syntactical compatibility with the standard PRINT statement. However, use of these functions in PRINT TO produces unreliable results.

Use of the TAB function is implemented as follows:

- Start of line is determined by scanning backward through the used portion of the buffer for a HEX(00).
- From that point, the buffer is scanned forward and the current tab position is determined by counting all characters greater than or equal to HEX(10).
- If the specified TAB position is greater than the current TAB counter as calculated above, sufficient spaces are appended to the end of the buffer to set the TAB position as specified.

If the specified TAB position is less than the current TAB position, no spaces are generated and the current location is unchanged; do not backspace in the buffer by use of TAB.

Use of a comma as a delimiter between items tabs to the next 16-character zone, based on the current TAB position. Use of the semicolon as a delimiter produces no actual output.

PRINT TO (cont.)

NOTE: The slight difference produced by printing HEX(0D) as a hex literal versus printing the HEX(0D) in a variable is not preserved by PRINT TO.

For example:

```
0010 SELECT PRINT 005(80)
0020 DIM L$256
0030 A$=HEX(0D)
0040 PRINT "X";HEX(0D);"Y"
0050 PRINT "A";A$;"B"
0060 L$=BIN(0,2)
0070 PRINT TO L$;"X;HEX(0D);"Y"
0080 PRINT TO L$;"A";A$;"B"
0090 PRINT STR(L$,3,VAL(L$,2));
```

PRINT overstrikes X with Y. PRINT TO does not.

Examples:

```
0010 PRINT TO Q1$();A$;TAB(10);B$;" ";D
0010 PRINT TO Q1$();"ABCDEF",X+Y-F
0010 PRINT TO X$;1,1*J
0010 PRINT TO A$;B$,1,2,3,4;
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

PRINT TO is not supported on the Wang 2200.

References:

PRINT
PRINT TAB
PRINTUSING TO

$PRINTER

General Form:

Form 1:

`$PRINTER = `*`alpha-expression`*

Form 2:

`alpha-receiver`` = $PRINTER`

Where:

`alpha-expression`` = length of 256 characters.`

Discussion:

This statement allows a NPL application program to examine or modify the current printer translation table. Form 1 allows the $PRINTER system variable to be modified. Form 2 allows the $PRINTER system variable to be examined.

The $PRINTER system variable contains the 256-byte printer translation table currently in effect. The printer translation table contains the character equivalents to be sent to the print device in place of the character received from the NPL program. Byte 1 contains the replacement character for HEX(00), byte 256 contains the replacement character for HEX(FF), etc.

NOTE: Printer translation is only used for print class devices where the XLA=Y clause is specified in the $DEVICE statement for that device. Refer to Section 7.8.6 for details.

For example:

```
10 DIM X$(256)1
20 X$()=$PRINTER       : REM PLACE CURRENT TABLE IN X$
30 PRINT STR(X$(),66,2): REM RESULTS WOULD BE "AB"
40 STR(X$(),66,1)="B"  : REM REPLACE CHARACTER IN POSITION 66
                         (VAL(HEX(41))+1) WITH THE CHARACTER "B"
50 $PRINTER=X$()       : REM MODIFY PRINTER TRANSLATION TABLE
```

The effect of this character is that whenever the character "A" is sent to the printer, the character "B" appears.

$PRINTER (cont.)

Changes made to the printer translation table using the $PRINTER statement go into effect immediately and remain in effect, unless further modifications are made or until the end of the current RunTime session. Changes are not retained from one session to the next. To make more permanent modifications to the printer translation table, please use the Printer Translation Table Editor Utility. However, when different printer translation values are required for different printers on the same system, use of the printer translation table may not be adequate. In these cases, the user of $PRINTER is required to set different values by the program when different printers are in use. Refer to Section 13.20 of the Programmer's Guide for further details.

NOTE: **The values of characters in the $PRINTER system variable refer to the character set of the native operating system. Available character sets vary from one machine to another. Refer to the appropriate NPL Supplement for details.**

Refer to Section 7.7.7 of the Programmer's Guide for further details on printer translation.

Examples:

Compatibility Issues:

$PRINTER statement is not a valid instruction in Wang 2200 Basic-2.

This statement is supported only with Release 2.01 or greater.

Default values for $PRINTER vary from one machine to another. Refer to the appropriate NPL Supplement for hardware-specific details.

References:

$DEVICE
Printer Translation Table Editor Utility - Section 13.20 of the Programmer's Guide
Printer Handling - Section 7.7 of the Programmer's Guide

PRINTUSING

General Form:

```
     PRINTUSING image-spec [{;}item]...[;]
                            {,}
```

Where:

```
     item       = {numeric-expression}
                  {alpha-variable    }
                  {literal-string    }

     image-spec = {literal-string specifying image-spec}
                  {alpha-variable containing image-spec}
                  {line-number of image-spec statement }
```

which can consist of one or more images separated by character strings where:

```
     image      = {[+] [$] [#[,]]...[.][#]...[^^^^] [+ ]}
                                                    [- ]
                                                    [- ]
                                                    [++]
                                                    [--]
```

Discussion:

The PRINTUSING statement is used to print output data according to a specially formatted output image to the currently selected PRINT device.

The image-spec defines the output format for items in the PRINTUSING statement. The image-spec is specified in one of three ways: as a literal-string, as an alpha-variable (which can be used by more than one PRINTUSING statement), or as a separate image-spec statement referenced by a line number (which can also be used by more than one PRINTUSING statement).

Individual print items are printed in the format specified by the corresponding image in the image-spec. That is, item #1 is formatted by the first image, item #2 by the second image, and so on. If more items are supplied than there are image locations, the image locations are reused from the beginning of the image-spec.

PRINTUSING (cont.)

The number of characters printed for an item always equals the number of characters in the image for that item.

Any character strings contained in the image are printed relative to the position of the images. That is, a character string preceding image1 prints before item #1, and a character string following item #1 prints after item #1. Character strings are printed only up to the end of the string following the last item printed. Strings following unused images are not printed.

The image controls the formatted output of numeric data as follows:

1. When using Integer format (###), if the value to be printed is shorter than the specified format, the output is padded with leading spaces. If the value is larger than the specified image, the image is printed in place of the value.

2. When using Fixed-Point format (###.##), both the integer and decimal part of the value are printed according to the image. If the integer value is shorter than the specified image, the output is padded with leading spaces. If the integer value is larger than the specified image, the image is printed in place of the value. The decimal part is either padded with trailing zeroes or truncated to fit the image.

3. When using Exponential format (###.#^^^^), the exponent is printed in the form "E±ee" where "ee" are the exponential digits. If the value is larger than the specified image, the image is printed in place of the value.

4. If the image begins or ends with a "+" or "-" sign, the proper sign of the value (positive or negative) is printed at the beginning or end of the output. A "+" prints a "+" or "-" and a "-" prints a "(blank)" or "-". If the image uses a leading sign, the sign is printed before the first digit.

5. If the image ends with a "++" or "--" sign, a "CR" or "DB", respectively, is placed at the end of the output for any negative values. Positive values end with two spaces.

6. If the image contains a dollar sign ($), the dollar sign is printed immediately preceding the first digit before the decimal, following the sign of the value (if a sign designator is specified at the beginning of the image).

PRINTUSING (cont.)

NOTE: The dollar sign ($) character in an image may be replaced with another character by setting byte 4 of the $OPTIONS system variable to the hex value of the desired character. Refer to the $OPTIONS statement for details.

7. If the image contains a comma (,) or decimal point (.), the character is printed in the corresponding output position if required to separate preceding and following digits.

NOTE: The comma (,) or decimal point (.) characters in an image may be replaced with another character by setting byte 5 or 6, respectively, of the $OPTIONS system variable to the hex value of the desired character. Refer to the $OPTIONS statement for details.

The image controls the formatted output of alpha-numeric data as follows:

1. Each image character is replaced by one item character from the alpha value. The alpha-item is left-justified in the image, and either extended with trailing blanks or truncated to fit the image. If the alpha-item is too long, it is right-truncated.

Example:
```
:0010 X$="##.#####^^^^"
    : Y$="############"
:0020 PRINTUSING X$,"Niakwa NPL"
    : PRINTUSING Y$,"Niakwa NPL"
:RUN
Niakwa NPL
Niakwa NPL
```

Suppression of Carriage Return

Normally when using a single-format specification to print more than one value, a carriage return is performed after each value is printed. The comma and semicolon delimiters may be used to control print output. A comma following an item causes a carriage return to be performed if the value uses the last image in the image-spec. A semicolon suppresses this carriage return and places consecutive items immediately after one another. At the end of the statement, a semicolon suppresses the start of a new line, therefore, PRINT or PRINTUSING output from subsequent statements would follow that of the current statement on the same output line.

PRINTUSING (cont.)

For example:

```
:0010 X$="#####"
:0020 PRINTUSING X$, 11; 12; 13
:RUN
   11   12   13
```

Examples:

```
0010 PRINTUSING "my name is #####";A$
0010 PRINTUSING X$,1234
0010 PRINTUSING 50,A,B,C,D
0050% $###,###.##   ######    ######   ###,###.##

:0010 %The balance in your account is $##,###.##
:0020 T=1234.56: R=54.39
:0030 PRINTUSING 10,T+R
:RUN
The balance in your account is $ 1,288.95

:0010 A$="TOTAL QUANTITY"
:0020 B$="PRODUCED"
:0030 C=91
:0040%THE ############## OF ITEMS ######## = ####
:0050 PRINTUSING 40,A$,B$,C
:RUN
 THE TOTAL QUANTITY OF ITEMS PRODUCED =   91

:0010 A$="PAY TO"
:0020 B$="BEARER"
:0030%##### ###### $#,###.##
:0040% FROM ##########
:0050 PRINTUSING 30,A$,B$,500;
:0060 PRINTUSING 40,"JOHN SMITH"
:RUN
PAY TO BEARER   $500.00 FROM JOHN SMITH
```

Compatibility Issues:

NPL allows the characters output from an image specification for the dollar sign ($), comma (,), and the decimal point (.) to be replaced by another character specified in the $OPTIONS system variable. These characters cannot be modified in Wang 2200 Basic-2.

References:

IMAGE
$OPTION

PRINTUSING TO

General Form:

```
PRINTUSING TO alpha-variable,image-spec [{;}item]...[;]
                                         {,}
```

Where:

```
item       = {numeric-expression}
             {alpha-variable    }
             {literal-string    }

image-spec = {literal-string specifying image-spec}
             {alpha-variable containing image-spec}
             {line-number of image statement      }
```

which can consist of one or more images separated by character strings where:

```
image      = {[+] [$] [#[,]]...[.][#]...[^^^^] [+ ]}
              [-]                              [- ]
                                               [++]
                                               [--]
```

Discussion:

The PRINTUSING TO statement formats a list of variables according to the referenced image-spec using the same logic as the PRINTUSING statement. However, instead of printing the result, it is stored in an alpha-variable for later use.

The first two bytes of the alpha-variable are used to store a count of characters (in binary) placed in the alpha-variable by the PRINTUSING TO statement. Whenever the PRINTUSING TO statement is executed, this count is retrieved to determine the starting location within the alpha-variable to begin storing output. The count is then updated to reflect the new count of characters in the alpha-variable. This allows multiple PRINTUSING TO operations to place successive buffered output in the correct location. PRINTUSING TO statements may be intermixed with PRINT TO statements referencing the same receiver-variable.

The count should be initialized to HEX(0000) before using an alpha-variable for the first time or before reusing an alpha-variable.

PRINTUSING TO (cont.)

If the total number of characters to be stored in the alpha-variable would exceed the length of the alpha-variable, output is truncated and the count is set to the length of the alpha-variable.

NOTE: **No character is inserted following carriage returns, and that carriage returns are not automatically generated when line width is exceeded.**

Examples:

```
0010 PRINTUSING TO Q1$(),"### ",A,B,C,D
0010 PRINTUSING TO Q1$(),8320,"ABCDEF",X
0010 PRINTUSING TO X$,"##",1,1*J
0010 PRINTUSING TO A$,B$,1,2,3,4;
```

Compatibility Issues:

Wang 2200 Basic-2 permits invalid initial values in the count characters of the receiving alpha-variable. NPL flags bad initial values with a runtime error.

References:

PRINTUSING
PRINT TO

PROCEDURE

General Form:

```
     PROCEDURE 'name [(parameter[,parameter]...)][attribute]...
```

Where:

```
     name      =  identifier


     paramerter = /POINTER] [_]variable [([dim1[,dim2]])][length]
                  [_]variable([dim1,[dim2]])

     attribute  = {/PUBLIC   }
                  {/FORWARD  }
                  {/EXTERNAL }
                  {/BEGINS   }
                  {/MAIN     }
                  {/EXIT     }
```

Discussion:

This statement declares the entry point of a named procedure, and the parameters to that procedure (if any). If the /FORWARD keyword is not specified, statements following the PROCEDURE statement define the body of the procedure. These must be followed by a matching END PROCEDURE statement.

For a discussion of PROCEDURE parameters, refer to "Common Properties of FUNCTIONs and PROCEDUREs", Section 4.8 of the Programmer's Guide.

Procedures declared with the attributes MAIN and EXIT are special purpose procedures used to ensure orderly initialization and shutdown of a module. At most one of these types of procedures may be declared in a module. The MAIN procedure (if any) is always executed first, whenever the module is resolved. The EXIT procedure is always executed last, whenever the module is about to be deresolved. Both the MAIN and EXIT procedures have no parameters.

PROCEDURE (cont.)

NOTE: If a library contains a /MAIN procedure, this is automatically called immediately after the module is resolved. This only occurs once. If the procedure is halted and cancelled or does a RETURN ERROR, NPL will not call the procedure automatically a second time.

Exiting the RunTime deresolves all modules, in order to ensure all /EXIT procedures are run.

Calling PROCEDUREs

Procedures have no return values and may not appear in expressions, but may be invoked by the statement 'identifier[(parameters)]. When the statement is executed, parameter values are passed and execution proceeds with the first executable statement in the function. Execution of a RETURN statement in the PROCEDURE body causes execution to proceed on the statement following the procedure call.

Calls to functions or procedures are permitted from immediate mode.

There is no implied HALT before functions or procedures are executed.

To step through the function, use the STEP mode.

If a function is called from immediate mode, then, even if the function is halted or STOPped for debugging, any immediate mode statements following the function are eventually executed when the function returns.

The scoping requirements for calls to functions from immediate mode are relaxed to make it easier to define and use "resident" modules of public functions. The normal scoping rules are first applied to any function name used in immediate mode. In immediate mode only, if the named function is not in scope, the public function list is then searched for a match.

Examples:

```
0010 PROCEDURE 'ProcessRecord
0010 PROCEDURE 'Initialize/MAIN
0010 PROCEDURE 'Shutdown/EXIT
0010 PROCEDURE 'MoveWindow(/POINTER Window_POS)
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

PROCEDURE (cont.)

References:

END PROCEDURE
CLEAR
Functions and Procedures - Section 4.8 of the NPL Programmer's Guide

’Procedure-name (Call PROCEDURE)

General Form:

```
’procedure-name[(argument[,argument]...)]
```

Where:

```
procedure-name = {Identifier       }
                 {<alpha-variable> }


argument       = {numeric-expression  }
                 {alpha-variable      }
                 {literal-string      }
```

Discussion:

PROCEDUREs may be called by specifying the Identifier preceded by "’" and followed by an argument list in parentheses (if required).

NOTE: FUNCTIONs have return values and may not be called in this way. They may only appear in expressions of the appropriate type.

Examples:

```
0010 ’Initialize
0010 ’ResetTerminal(_WHITE,_BRIGHT_CYAN,3,HEX(020D0C030E))
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

Refer to Section 4.8 of the Programmer’s Guide for more information.

$PROGRAM

General Form:

```
alpha-receiver = $PROGRAM
```

Discussion:

The $PROGRAM system variable contains the same text string that is returned by the Program Load Sequence section of LISTDT. This text string contains the names of up to 6 program modules currently residing in partition memory. The first program name is the first program loaded since the most recent CLEAR or LOAD RUN statement. The next 5 program names are the last programs loaded since the most recent CLEAR or LOAD RUN statement. The text string may also contain certain indicators:

The value returned by $PROGRAM only shows the program loaded or overlaid into the RUN module. Modules loaded using INCLUDE directives do not affect the value of $PROGRAM.

+ indicates that following program was the next module loaded.

... indicates that there were modules loaded that are not listed (because the total number of modules in memory exceeds 6).

The information returned by $PROGRAM is updated only by LOAD, LOAD RUN, and LOAD BOOT statements or commands. It is not updated by LOAD DA.

$PROGRAM cannot appear on the left side of an equivalence statement. $PROGRAM is intended to provide information only. Typical use of $PROGRAM would be to display a message to the operator in the event of certain types of errors.

Examples:

```
0010 X$=$PROGRAM
0010 Y$()=$PROGRAM
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

This statement is not supported on the Wang 2200.

$PROGRAM (cont.)

References:

LISTDT

$PSTAT

General Form:

Form 1:

```
$PSTAT = alpha-expression
```

Form 2:

```
alpha-receiver = $PSTAT(numeric-expression)
```

Discussion:

The $PSTAT statement is a special instruction which returns various status information for the partition specified in the expression. The argument for the function must be specified as a numeric-expression equal to the partition number. $PSTAT contains eight bytes that can be set by form 1 of the statement and accessed by the user by using form 2 of the $PSTAT statement.

An alpha-variable can be set to the contents of $PSTAT. This variable contains the following:

Byte	Contents
1-8	User-defined area.
9	Operating system type (always "C", for NPL).
10	RunTime revision number. Acceptable values are: Revision 3.00.xx - HEX(30) Revision 2.01.xx - HEX(21) Revision 2.00.xx - HEX(20) Revision 1.03 - HEX(13).
11	Since global partitioning is not supported in NPL, the Bank # is set to #PART in packed (##) format.
12-13	Always contains SPACEK in packed (##.##) format. If SPACEK>99, this field contains 99.00 (in packed decimal format).
14	Programmability ("P" or " ").
15	Terminal number in packed (##) format.

$PSTAT (cont.)

Byte	Contents
16	Partition status ("A" for active, "D" for partitions in background that are not waiting for a terminal to be attached, "W" for partitions in background that are waiting for a terminal to be attached).
17-24	Blank.
25	ERR function value (numeric portion of the last error encountered (in Hex)).
26-28	These 3 bytes indicate the following partition # assignments, respectively: Text (containing program text), Global (containing global operations), and DATA (containing DATA statements referenced by a current READ). Since global partitioning is not supported in NPL, this only contains #PART in packed (##) format.
29	Device-address (address of device which the partition is currently using or which it is currently waiting).

Examples:

```
0010 Q$ = $PSTAT(#PART)
0010 B$(),STR(A$,1,30) = $PSTAT(#PART)
```

Compatibility Issues:

Information retained by $PSTAT is different than on a Wang 2200. However, it is consistent with practical use of $PSTAT.

Generation of background partition status (byte 16) is a feature that is supported only on NPL Revision 3.0 or greater and is highly operating system-specific. Refer to the NPL Supplement for details on background partition support on the operating system.

References:

PUBLIC

General Form:

```
PUBLIC [PackageIdentifier]
```

Discussion:

The PUBLIC statement specifies the start of a set of statements which define the interface to publicly visible components of a module. The PUBLIC section includes all statements up to a matching END PUBLIC statement, which is required.

If a PackageIdentifier is not specified, the PUBLIC section defines the default interface to the module, which is automatically USED by any module which references the module using an INCLUDE statement.

All statements are permitted in a PUBLIC section; however, only certain declarative statements become "visible" to other modules which reference the interface section (using a USES statement). In particular:

- DIM /PUBLIC variable declarations
- RECORD /PUBLIC record declarations
- FUNCTION /PUBLIC function prototypes (usually FORWARD)
- PROCEDURE /PUBLIC function prototypes (usually FORWARD)
- DEFFN /PUBLIC marked subroutine declarations (usually FORWARD)
- USES statements
- INCLUDE statements

PUBLIC (cont.)

NOTE: **For each of the above statements, the /PUBLIC attribute is implied when the statement occurs within a PUBLIC section.**

Consequently, it is good programming practice to restrict statements within the PUBLIC section to the above statements and any relevant comments.

A module may contain more than one PUBLIC statement, and these may be nested.

USES and INCLUDE statements should only be part of a PUBLIC section if it is necessary for information referenced by these statements to be treated as if it is part of the interface to the module.

NOTE: **The declaration statements for PUBLIC items are visible to NPL programmers using the LIST PUBLIC statements, even if they appear in a library module which is scramble-protected. Proprietary or sensitive information should never be placed within a PUBLIC section.**

The PackageIdentifier, if used, must be unique within the workspace. PUBLIC entities within a PUBLIC section where PackageIdentifier is used are visible only to other modules that execute a USES statement within a matching PackageIdentifier.

Examples:

```
0010 PUBLIC
0010 PUBLIC StringFunctions
0010 PUBLIC StandardColorNames
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

END PUBLIC
INCLUDE
USES
Program Modules - Section 4.10 of the NPL Programmer's Guide

READ

General Form:

```
     READ variable [,variable]...
```

Where:

```
     variable = {numeric-receiver}
                {alpha-variable  }
```

Discussion:

The READ statement is used to assign a list of values from a DATA statement(s) to a list of variables in the READ statement. The process uses the data values sequentially, beginning with the DATA statement with the lowest line number, using each value on that statement, then those (in order) from the next DATA statement in the program. An error occurs if a READ is executed and the list has been exhausted. An error also occurs if the value being transferred does not match the variable type required by the list.

As each value from a DATA statement is used by a READ statement, a DATA pointer is incremented by one. DATA values may be reused through use of the RESTORE statement, which also allows for setting the DATA pointer to a specified value within a particular DATA statement. Refer to the discussion on RESTORE in this section.

Each module has its own separate list of DATA items. However, all modules share a single global data pointer.

The data pointer is set by default to the start of a module when resolution of that module completes. A RUN statement will set the data pointer to the start of the RUN module's DATA list. Data items in other modules may not be read, until a RESTORE statement is executed in the other module.

Examples:

```
0010 READ A,B,STR(C$(),1,10)
0010 READ B,C,D,E
0010 READ A$,B
0010 READ C(1,2),C(1,3),D$(2)

:0010 READ A,B,C$,D
:0020 PRINT A;B;C$;D
```

```
:0030 DATA 1,2,"NPL",3,34,"ABC",1234
:RUN
 1  2 NPL 3
```

READ (cont.)

Compatibility Issues:

References:

DATA
RESTORE

READ DC

General Form:

```
     READ DC T [device-address,  ] alpha-receiver, numeric-receiver
               [disk-address,    ]        [,restrict] ...
               [<alpha-variable>,]
```

Where:

```
     restrict   = {[FILE  rel-op] alpha-mask         }
                  { TYPE  rel-op  alpha-mask         }
                  { START rel-op  numeric-expression }
                  { END   rel-op  numeric-expression }
                  { USED  rel-op  numeric-expression }
                  { FREE  rel-op  numeric-expression }
                  { DATE  rel-op  alpha-mask         }
                  { TIME  rel-op  alpha-mask         }

     rel-op     = relational operator {<,=,>,<=,>=,<>}.

     alpha-mask = alpha-variable or alpha-literal.
```

Discussion:

The READ DC statement scans the file names contained in the index area of a specified diskimage, starting after the filename slot number indicated by the specified numeric-variable, and matches filenames found against specified restrictions. The first filename found is returned in the specified alpha-variable and the specified numeric-variable is set to the number of the filename slot where the filename was found. This allows subsequent iterations of READ DC to find the next matching filename. If no filename meeting the restriction requirements is located, the alpha-receiver is set to spaces and the numeric-receiver is set to zero.

There are 16 filename slots per index sector (the first slot of the first sector is reserved for diskimage level parameters).

READ DC (cont.)

READ DC is intended to be used as an alternative to direct access to the index area for applications which need to locate filenames without knowing the exact name(s) of the files. Typical use of READ DC requires that READ DC be executed within a loop with the numeric-variable set to zero before starting the loop (refer to example below). After the value of the numeric-receiver is set to zero outside of the loop, it should not be set again. Rather, it should just be examined after each iteration for a returned value of zero which indicates no files found.

Filenames are returned regardless of their status (unless the TYPE restriction is specified). The programmer may wish to use the LIMITS statement on filenames returned in order to access other information about the file including type, status, and sector locations.

Specifying Files

The READ DC command allows restriction of the file scan by specification of key words related to information about the file followed by a relational operator followed by a mask. As the catalog index is read, file parameters are matched against the specified mask as required by the relational operator. Only files meeting the specified requirements are returned by READ DC. Multiple restrictions may be specified, in which case only files meeting all requirements are returned by READ DC.

For keywords which represent alpha data, the mask must be a literal or alpha-variable. For keywords which represent numeric data, the mask must be a valid numeric expression of which the integer portion is used.

For alpha masks, standard wildcard usage is supported. That is, a "?" in any position matches any character in that position. An "*" indicates that any characters from the position of the asterisk to the end of the field match.

The key words available for file specification are:

FILE	Eight byte alpha
TYPE	Two byte alpha. Byte one is "S" if the file is scratched; blank if not scratched. Byte 2 is "P" for program files; "D" for data files.
START	Numeric

READ DC (cont.)

USED	Numeric
FREE	Numeric
DATE	8-byte alpha in the format yy/mm/dd
TIME	8-byte alpha in the format hh:mm:ss

NOTE: If no keyword is specified, the keyword FILE and the relational operator = are assumed.

Examples:

```
0010 READ DCT/D35,A$,A,"2C*"
0010 READ DCT<D$>,B$,X(1),TYPE="SP"

:0010 DIM A$8
:0020 A=0                          :REM Start at filename slot number zero
:0030 READ DCT/D35,A$,A,FILE="2C*" :REM Find files that start with "2C"
:0040 IF A=0 THEN 100              :REM No more matching files
:0050 PRINT A$,A
:0060 GOTO 30
:0100 STOP "DONE"
:RUN

2CCOPY            25
2CRCVR            26
2CBCKP            38
2CMENU            51
2CMNDATA          52
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

READ DC is not supported on the Wang 2200.

References:

LISTDC
LIMITS

RECORD

General Form:

```
RECORD [/PUBLIC] record-identifier
       [/STATIC]
```

Discussion:

The RECORD statement declares a record identifier with the given name. The statement begins the body of a logical record containing one or more fields declared with the FIELD statement. The complete record declaration ends with an END RECORD statement.

Records serve as templates for an abstract data type. The number of bytes required by an instance of the record is returned by the #RECORDLENGTH(record-identifier) built-in function. To declare a buffer for the record, declare a string variable with a length at least this large. This string variable may then be used as a buffer for the record.

NPL does not perform type-checking when field identifiers are attached to buffers. Any field name may be attached to any alpha-variable. NPL treats the alpha-variable as a buffer for the record to which the field identifier belongs.

All field-identifiers must be unique within the scope (STATIC/ PUBLIC) specified.

A RECORD identifier which occurs within a PUBLIC section is PUBLIC by default.

A RECORD statement may not occur within the body of another record specification.

Examples:

The following is an example of valid syntax.

```
0010 RECORD Payroll
0010 RECORD /PUBLIC Employee
0010 RECORD /STATIC Passwords
```

RECORD (cont.)

The following is a practical example of statement usage.

```
0010 RECORD Book_Record
   :   FIELD Title$30
   :   FIELD Author$30
   :   FIELD Publisher$30
   :   FIELD ISBN$30
   :   FIELD Edition = HEX(B004)
   : END RECORD
0020 DIM book_stats$#RECORDLENGTH(Book_Record)
   : book_stats$.Title$ = "Mouse In The House"
   : book_stats$.Author$ = "Dr. Seuss"
   : book_stats$.Edition = 4
   : book_stats$.ISBN$ = "0-12-011818-4"
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

FIELD
END RECORD
#RECORDLENGTH

#RECORDLENGTH Function

General Form:

```
#RECORDLENGTH (record-identifier)
```

Discussion:

The #RECORDLENGTH intrinsic function returns the record length in bytes for a given record identifier. This length can then be used to dimension an alpha-variable to be used as a record buffer for referencing and manipulating fields associated with the record-identifier. The use of the #RECORDLENGTH function in expressions used by declarations should only occur after the END RECORD statement for the record.

Examples:

```
0010 DIM PayrollRecord$#RECORDLENGTH(Payroll)
0010 DIM EmployeeRecord$#RECORDLENGTH(Employee)
0010 DIM PasswordsRecord$#RECORDLENGTH(Passwords)
0010 G=#RECORDLENGTH(BoxInfo)
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

END RECORD
RECORD

$RELEASE PART

General Form:

```
$RELEASE PART
```

NOTE: This statement is supported for Wang compatibility reasons only and its use in new development is not recommended.

Discussion:

The syntax of this statement is supported only for compatibility with Wang 2200 Basic-2. When executed under NPL, $RELEASE PART performs no operation.

Examples:

Compatibility Issues:

In Wang 2200 Basic-2, the $RELEASE PART instruction can be used to relinquish control of the current partition $RELEASE PART is executed from. NPL does not support operation of the $RELEASE PART instruction.

References:

$RELEASE TERMINAL

General Form:

```
     $RELEASE TERMINAL [TO partition-number [,STOP]]
```

Where:

```
     partition-number = a numeric expression containing the parti-
                        tion number to assign to foreground.
```

Discussion:

$RELEASE TERMINAL detaches the terminal from the current foreground partition and attaches the terminal to a background partition as follows:

1. If no background partition is present, $RELEASE TERMINAL performs no operation.
2. If no specific partition is specified in the TO clause, the terminal is attached to the next higher-numbered partition assigned to that terminal that is waiting for the terminal (where byte 16 of $PSTAT = "W"). If no higher-numbered partitions assigned to the same terminal are waiting, the lowest-numbered waiting partition is activated.
3. If the TO clause is specified with a partition-number, the terminal is attached to that specific partition. If the partition-number specified is not assigned to the terminal or is not waiting for the terminal (where byte 16 of $PSTAT = "W"), an X77 (Invalid Partition Reference) error results.

The STOP clause indicates that processing should be halted once the background partition is attached. Since the background partition can only be attached when it is waiting for keyboard input, the STOP clause performs no operation in NPL.

Examples:

```
10 $RELEASE TERMINAL TO 10
10 $RELEASE TERMINAL BACKGROUND
```

$RELEASE TERMINAL (cont.)

Compatibility Issues:

$RELEASE TERMINAL performs no operation on all revisions of NPL prior to Revision 3.0.

On the Wang 2200, $RELEASE TERMINAL TO partition-number succeeds even if the specified background partition is not waiting for the terminal.

Functionality of $RELEASE TERMINAL is extremely operating system dependent. On some operating systems, background partitions are not supported at all. On other operating systems, various limitations may be present. Refer to the NPL Supplements for details about the availability and functionality of background partitions on the operating system.

References:

$IF
$PSTAT

REM

General Form:

```
REM [%[^]] [text]
```

Discussion:

The REM statement is used to insert comments into a NPL program. Any characters except a colon may be used. A colon or end of program line terminates the REM statement.

The optional % parameter causes text within the REM to be listed on a separate line when the LIST D command is used. In addition, if the output device is the screen, text are printed bright. This allows a convenient way of adding program headings to program listings.

The optional "^" parameter causes a top-of-form to be performed to the printer before a program listing is performed when using the LISTD command.

NOTE: The "^" displays as an up-arrow (HEX(5E)) on standard displays.

REM statements are retained only if the $KEEPREMS system variable is set to HEX(01) or HEX(02), or by the compiler if KEEPREMS ON option is specified.

LINE REMs allow remark statements to include colons. This is particularly useful for developers who wish to maintain source code in ASCII format and use third-party source code control programs. LINE REMs are specified by a semicolon (";") as the first character in a statement. LINE REMs are terminated only by a soft carriage return or by the end of the program line. LINE REMs are NOT terminated by a colon.

For example:

```
0010 ; This is a remark. It can have a : embedded in it<
   :PRINT "THIS IS EXECUTABLE"<
   :;    This is another remark: This is still part of the remark.
          The remark can span multiple physical lines. It is terminated
          by a soft carriage return<
   : PRINT "THIS IS ALSO EXECUTABLE"
```

REM (cont.)

WARNING--The LINE REM and the IMAGE statement are the only statements that are affected by soft carriage returns. Care must be taken--deleting a soft carriage return following a LINE REM makes the subsequent statement non-executable (it is treated as part of the remark). For further information on soft carriage returns (sometimes referred to as "Return Graphics"), refer to Section 5.4 of the NPL Programmer's Guide.

All batch compiler (B2C) options that apply to standard REMs also apply to LINE REMs. In addition, the $PC clause (REM $PC) and the $KEEPREMS system variable apply to LINE REMs as they do to REM statements.

Examples:

```
0010 REM "START" Start-up Program
0010 REM
0010 REM This program updates the file.
0010 REM %^ Test Program BOOT1
0010 REM % Last Modification - 7/24/86
```

Compatibility Issues:

In Wang 2200 Basic-2, REM% causes text to print in expanded format when sent to the printer.

The REM statement is implemented in Revision 2.00 and greater of NPL. Prior revisions of the RunTime Program cannot load modules containing compiled REMs. Compilers for previous revisions removed REM statements.

References:

REM $PC

REM $PC

General Form:

```
     REM $PC statement
```

NOTE: This statement is supported for Wang compatibility reasons only and its use in new development is not recommended.

Discussion:

Statements which follow the special designation of REM $PC are executable in NPL. The REM $PC statement allows programs written for the Wang 2200 to be upgraded to make use of extensions added for the NPL version, yet while maintaining compatibility with the 2200 system. This is done by coding statements whose syntax is not 2200 compatible within special types of REM (remark) statements. Since all REM statements are ignored on the Wang 2200, the program code in these REM statements does not cause difficulties on the 2200.

Under the interpretive RunTime Program, setting the $KEEPREMS system variable to hex(00) causes the REM $PC prefix to be removed upon entering the line. For example, the following statement:

```
0010 REM $PC $HELP="ARCODE"
```

with $KEEPREMS=HEX(00) would actually be stored as:

```
0010 $HELP="APCODE"
```

Refer to the $KEEPREMS system variable for details.

Using the REM$ ON compiler option, the NPL compiler can be instructed to compile statements within REM $PC statements for use under NPL. Refer to Chapter 9, Compiler Options, of the NPL Supplements for details on the format of compiler options.

REM $PC (cont.)

Examples:

```
10 REM $PC $HELP="ARCODE"
:  REM $PC PRINT "Help information is available."
:  LINPUT "ENTER THE APPROPRIATE A/R CODE",-C$
```

The example shows how on-line help screens might be added to a Wang 2200-compatible program. The $HELP statement (which is not a Wang 2200 Basic-2 instruction) executes only in NPL. The PRINT statement executes only in NPL.

Comments and Cautions:

It is recommended that the use of the REM $PC statement be limited. In particular, be aware of the following potential problems:

- Since references to LINE numbers within a REM $PC are not be RENUMBERed by the Wang 2200, avoid using them within a REM $PC.
- Colons (":") within quote strings should not appear in REM $PC statements.
- Additional software testing time on the Wang 2200 should be allocated to ensure that no errors were introduced by REM $PC statements on the NPL version.

Compatibility Issues:

REM $PC statements are functionally the same as REM statements in Wang 2200 Basic-2.

Revisions prior to Revision 2.00 of NPL allowed the "PC" portion of REM $PC to be optional for some statements.

References:

$KEEPREMS
REM

RENAME

General Form:

```
RENAME T [file-number,  ] old-file-name TO new-file-name
         [disk-address, ]
         [<address-var>,]
```

Where:

```
old-file-name = an alpha-variable or literal-string containing
                the old name of the file.

new-file-name = an alpha-variable or literal string containing
                the new name of the file.
```

Discussion:

RENAME changes the name of an existing file (old-file-name) to the specified new-file-name. The contents of the file are not affected. Only the diskimage catalog index is affected. The specified old-file-name may be either scratched or not scratched and may be either a program or a data file. The file type of the new-file-name is identical to the old-file-name.

Once a file has been successfully RENAMEd, it may no longer be accessed by its old name.

If the new-file-name already exists, or if the index is full, an error occurs and the index entry for the old-file-name remains unmodified.

Examples:

```
0010 RENAME T/D11,"PROG1" TO "PROGA"
0010 RENAME T#Y,A$ TO B$
:RENAME T"OLDPROG" TO "NEWPROG"
:RENAME T<A$>,"ARDATA TO ARDATASV"
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

References:

RENAME DEFFN’

General Form:

```
RENAME DEFFN’ old-deffn-name [MERGE] TO new-deffn-name
```

Where:

```
old-deffn-name  = current name of a DEFFN’ subroutine in
                  the program.

new-deffn-name =  name to which old-deffn-name references
                  should be changed.
```

Discussion:

RENAME DEFFN’ is an immediate mode command that is used to RENAME all occurrences of a DEFFN’ identifier in the currently selected LIST module to a new or existing DEFFN’ identifier.

For example:

```
RENAME DEFFN ’Get_Blue TO ’Get_Red
```

renames all occurrences of the DEFFN identifier ’Get_Blue in the current LIST module to DEFFN’ identifier ’Get_Red.

NOTE: RENAME DEFFN may not be used to change the value of numbered DEFFN’s. Only named DEFFN’s are permitted.

The prime (’) indicator is optional, but is always tagged by the decompiler.

If the old DEFFN’ name does not exist, an error 202 "Name Not Referenced in Program" occurs.

If the new DEFFN’ name already exists, an error 201 "Name Already Referenced in Program" occurs unless, the keyword MERGE is specified.

RENAME DEFFN’ (cont.)

For example:

```
RENAME DEFFN ’Grape_Juice MERGE TO ’Vintage_Stuff
```

renames all occurrences of the DEFFN identifier ’Grape_Juice in the current LIST module to the DEFFN identifier ’Vintage_Stuff, even if ’Vintage_Stuff has been previously defined in the current LIST module.

Renaming DEFFN’ identifiers deresolves the LIST module and clears the return stack.

NOTE: A HALTed program may not be CONTINUEd after any RENAME DEFFN’ statement.

RENAME DEFFN’ may not be used under program control.

Examples:

```
RENAME DEFFN ’Old_Pack TO ’New_Pack
RENAME DEFFN  Count_Sticks TO Count_Trees
RENAME DEFFN ’Aspirin$ MERGE TO ’Ibuprofen$
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

$NAMEOF
DEFFN’

RENAME FIELD

General Form:

```
RENAME FIELD .old-field-name [$] [(] [MERGE] TO .new-field-name [$] [(]
```

Where:

old-field-name	= current name of a field identifier in the program.
new-field-name	= name to which old-field-name should be changed.

Discussion:

RENAME FIELD is an immediate mode command that is used to RENAME all occurrences of a FIELD identifier in the currently selected LIST module to a new or existing FIELD identifier.

For example:

```
RENAME FIELD .Blue TO .Red
```

renames all occurrences of the FIELD identifier .Blue in the current LIST module to FIELD identifier .Red.

The type information for string fields "$" and arrays "(" must be specified. The types of the old and new FIELD identifiers must be the same.

The (.) indicator is optional, but is always tagged by the decompiler.

If the old field name does not exist, an error 202 (Name Not Referenced in Program) occurs.

If the new field name already exists, an error 201 (Name Already Referenced in Program) occurs, unless the keyword MERGE is specified.

RENAME FIELD (cont.)

For example:

```
RENAME FIELD .Grape( MERGE TO .Raisin(
```

renames all occurrences of the FIELD identifier .Grape(in the current LIST module to the FIELD identifier .Raisin(even if .Raisin(has been previously defined in the current LIST module.

Renaming FIELD identifiers deresolves the LIST module and clears the return stack.

NOTE: A HALTed program may not be CONTINUEd after any RENAME FIELD statement.

RENAME FIELD may not be used under program control.

Examples:

```
RENAME FIELD .Particle TO .Quark
RENAME FIELD .Negative$ TO .Positive$
RENAME FIELD .Aspirin$ MERGE TO .Ibuprofen$
RENAME FIELD .Stripe$( MERGE TO .Shirt$(
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

$NAMEOF

RENAME FUNCTION

General Form:

```
RENAME FUNCTION 'old-function-name [$] [MERGE] TO 'new-function-name [$]
```

Where:

old-function-name	= current name of a FUNCTION identifier in the program.
new-function-name	= name to which the FUNCTION identifier should be changed.

Discussion:

RENAME FUNCTION is an immediate mode command that is used to RENAME all occurrences of a FUNCTION identifier in the currently selected LIST module to a new or existing FUNCTION identifier.

For example:

```
RENAME FUNCTION 'Red TO 'Blue
```

renames all occurrences of the FUNCTION identifier 'Red in the current LIST module to FUNCTION identifier 'Blue.

The type information for string functions "$" must be specified. The types of the old and new FUNCTION identifiers must be the same. The prime (') indicator is optional, but is always tagged by the decompiler.

If the old function name does not exist, an error 202 (Name Not Referenced in Program) occurs.

If the new function name already exists, an error 201 (Name Already Referenced in Program) occurs unless, the keyword MERGE is specified.

RENAME FUNCTION (cont.)

For example:

```
RENAME FUNCTION ’Grape_Juice MERGE TO ’Vintage_Stuff
```

renames all occurrences of FUNCTION ’Grape_Juice in the current LIST module to ’Vintage_Stuff even if ’Vintage_Stuff has been previously defined in the current LIST module.

Renaming FUNCTION identifiers deresolves the LIST module and clears the return stack.

NOTE: A HALTed program may not be CONTINUEd after and RENAME FUNCTION statement

RENAME FUNCTION may not be used under program control.

```
RENAME FUNCTION ’Do_It$ TO ’Did_It$
RENAME FUNCTION ’Count_Sticks TO ’Count_Trees
RENAME FUNCTION ’Aspirin$ MERGE TO ’Ibuprofen$
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

$NAMEOF
RENAME PROCEDURE

RENAME PROCEDURE

General Form:

```
RENAME PROCEDURE 'old-procedure-name [MERGE] TO 'new-procedure-name
```

Where:

```
old-procedure-name        = current name of a PROCEDURE in the
                            program.

new-procedure-name        = name to which old-procedure-name
                            should be changed.
```

Discussion:

RENAME PROCEDURE is an immediate mode command that is used to RENAME all occurrences of a PROCEDURE identifier in the currently selected LIST module to a new or existing PROCEDURE identifier. For example:

```
RENAME PROCEDURE 'Get_Blue TO 'Get_Red
```

renames all occurrences of the PROCEDURE identifier 'Get_Blue in the current LIST module to PROCEDURE identifier 'Get_Red.

The prime (') indicator is optional, but is always tagged by the decompiler.

If the old procedure name does not exist, an error 202 "Name Not Referenced in Program" occurs.

If the new procedure name already exists, an error 201 "Name Already Referenced in Program" occurs unless, the keyword MERGE is specified.

For example:

```
RENAME PROCEDURE 'Grape_Juice MERGE TO 'Vintage_Stuff
```

renames all occurrences of the PROCEDURE identifier 'Grape_Juice in the current LIST module to the PROCEDURE identifier 'Vintage_Stuff even if 'Vintage_Stuff has been previously defined in the current LIST module.

RENAME PROCEDURE (cont.)

Renaming PROCEDURE identifiers deresolves the LIST module and clears the return stack.

NOTE: A HALTed program may not be CONTINUEd after any RENAME PROCEDURE statement.

RENAME PROCEDURE may not be used under program control.

Examples:

```
RENAME PROCEDURE ’Do_It TO ’Did_It
RENAME PROCEDURE ’Count_Sticks TO ’Count_Trees
RENAME PROCEDURE ’Aspirin MERGE TO ’Ibuprofen
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

$NAMEOF
RENAME FUNCTION

RENAME RECORD

General Form:

```
RENAME RECORD old-record-name  [MERGE] TO new-record-name
```

Where:

old-record-name	= current name of a RECORD identifier in the program.
new-record-name	= name to which old-record-name should be changed.

Discussion:

RENAME RECORD is an immediate mode command that is used to RENAME all occurrences of a RECORD identifier in the currently selected LIST module to a new or existing RECORD identifier.

For example:

```
RENAME RECORD Colors TO Hues
```

renames all occurrences of the RECORD identifier Colors in the current LIST module to RECORD identifier Hues.

If the old record name does not exist, an error 202 "Name Not Referenced in Program" occurs.

If the new record name already exists, an error 201 "Name Already Referenced in Program" occurs unless, the keyword MERGE is specified.

For example:

```
RENAME RECORD Grapes MERGE TO Raisins
```

renames all occurrences of the RECORD identifier Grapes in the current LIST module to the RECORD identifier Raisins even if Raisins has been previously defined in the current LIST module.

RENAME RECORD (cont.)

Renaming RECORD identifiers deresolves the LIST module and clears the return stack.

NOTE: A HALTed program may not be CONTINUEd after any RENAME RECORD statement.

RENAME RECORD may not be used under program control.

Examples:

```
RENAME RECORD Particles TO Quarks
RENAME RECORD Negatives TO Positives
RENAME RECORD Aspirin MERGE TO Ibuprofen
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

$NAMEOF

RENAME = (Statement Label)

General Form:

```
    RENAME  =  old-statement-label [MERGE] TO new-statement-label
```

Where:

old-statement-label	= current name of a statement label in the program.
new-statement-label	= name to which old-statement-label should be changed.

Discussion:

RENAME = is an immediate mode command that is used to RENAME all occurrences of a statement label in the currently selected LIST module to a new or existing statement label.

For example:

```
RENAME = Get_Color TO Get_Background
```

renames all occurrences of the statement label Get_Color in the current LIST module to statement label Get_BackGround.

If the old statement label name does not exist, an error 202 (Name Not Referenced in Program) occurs.

If the new statement label name already exists, an error 201 (Name Already Referenced in Program) occurs unless, the keyword MERGE is specified.

For example:

```
RENAME = Make_Wine MERGE TO Brew_Beer
```

renames all occurrences of the statement label Make_Wine in the current LIST module to the statement label Brew_Beer even if Brew_Beer has been previously defined in the current LIST module.

Renaming statement labels deresolves the LIST module and clears the return stack.

RENAME = (Statement Label) (cont.)

NOTE: A halted program may not be CONTINUEd after any RENAME = statement.

RENAME = may not be used under program control.

Examples:

```
RENAME = Under_Control TO OutOfControl
RENAME = Calc_Principal TO Calc_Interest
RENAME = Get_Color MERGE TO Get_BackGround
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

RENAME V

General Form:

```
RENAME V  old-variable-name [MERGE] TO new-variable-name
```

Where:

```
old-variable-name        = {numeric scalar          }
                           {numeric array           }
                           {alpha scalar            }
                           {alpha array             }

new-variable-name        = {numeric scalar          }
                           {numeric array           }
                           {alpha scalar            }
                           {alpha array             }
```

Discussion:

RENAME V is an immediate mode command that is used to RENAME all occurrences of a variable in the currently selected module to a new or existing variable.

For example:

```
RENAME V A$ TO B$
```

renames all occurrences of variable A$ currently in memory to B$.

Variable names must be of the same type (numeric scalar, numeric array, alphanumeric scalar, or alphanumeric array).

If the old variable name does not exist, an error 202 (Variable Already Referenced in Program) occurs.

If the new variable name already exists, an error 201 (Variable Already Referenced in Program) occurs unless the keyword MERGE is specified.

RENAME V (cont.)

For example:

```
RENAME V A$ MERGE TO B$
```

renames all occurrences of variable A$ in the current LIST module to B$, even if B$ is previously defined in the current LIST module.

RENAME V implicitly performs a CLEAR V if either the new-name or old-name is a common (COM) variable.

RENAME V implicitly performs a CLEAR N if either the new-name or old-name is a declared (DIM) variable, and neither is common.

RENAME V always deresolves the LIST module. Program execution may not be continued after a RENAME V command.

RENAME V may not be used under program control.

Examples:

```
RENAME V A$ TO X43$
RENAME V Name$( MERGE TO OldName$(
RENAME V Apples MERGE TO Oranges
RENAME V Grapes( TO Raisins(
```

Compatibility Issues:

This statement is supported only with Release 3.2 or greater.

References

RENUMBER

General Form:

```
    RENUMBER [line1][-line2] [TO line3] [STEP x]
```

Where:

```
    line1 = first line-number to be renumbered.  If omitted, the low-
            est line-number in the program is assumed.

    line2 = last line-number to be renumbered.  If omitted, the high-
            est line-number in the program is assumed.

    line3 = new starting line-number.  If omitted, the default is the
            same as the STEP value.

    x     = STEP value, an integer such that 0< x <100 (default=10).
```

Discussion:

The RENUMBER command is used to renumber an entire program or portions of a program by any specified incremental STEP value. RENUMBER can also be used to reorder lines within a program, by moving sections of program text from one location to another, and appropriately changing the corresponding line numbers. RENUMBER also performs a cross-reference of the program to check for line references, changing any references to the new line number.

The lines which are in the line-number range are always consecutive before the RENUMBER, and must be consecutive after the RENUMBER or an error P33-Line number conflict is generated and the operation is not performed.

Renumbering a program allows subroutines or program segments to be inserted within a program without reentering the entire program due to insufficient line numbers

The RENUMBER statement performs no operation on the non-interpretive RunTime program.

RENUMBER (cont.)

Examples:

```
:RENUMBER
:RENUMBER -1000 TO 10
:RENUMBER 200-1000 TO 100 STEP 5
:RENUMBER STEP 5
```

Given the following program in memory:

```
0050 GOSUB 1000
0100 PRINT "ABC"
0200 PRINT "123"
0210 I=I+1
0300 PRINT "XYZ"
0350 IF I=12 THEN 200
0400 GOSUB 2000
```

Using the RENUMBER function to renumber this program segment from line 100 to line 350:

```
:RENUMBER 100-350 TO 100 STEP 10
:LIST
0050 GOSUB 1000
0100 PRINT "ABC"
0110 PRINT "123"
0120 I=I+1
0130 PRINT "XYZ"
0140 IF I=12 THEN 110
0400 GOSUB 2000
```

Given the following program in memory:

```
0100 DIM A$20,B$50
0110 DIM R(4),T(5,5)
0200 GOSUB 1000
0210 GOSUB 2000
0500 FOR I=1 TO 100 STEP 2
0510 PRINT J,K,L
0520 GOSUB 3000
0530 NEXT I
0540 A$=W$(9)
```

RENUMBER (cont.)

Using RENUMBER to reorder a section of program lines:

```
:RENUMBER 500-530 to 150 STEP 5
:LIST
0100 DIM A$20,B$50
0110 DIM R(4),T(5,5)
0150 FOR I=1 TO 100 STEP 2
0155 PRINT J,K,L
0160 GOSUB 3000
0165 NEXT I

0200 GOSUB 1000
0210 GOSUB 2000
0540 A$=W$(9)
```

Compatibility Issues:

This statement is supported only with Release 2.0 or greater.

References:

REPEAT

General Form:

REPEAT

Discussion:

The REPEAT statement marks the start of a structured REPEAT...UNTIL loop. It may be followed by a number of statements, which comprise the body of the loop. It must then be followed by an UNTIL statement.

The body of a REPEAT...UNTIL loop is always executed at least once. When a REPEAT statement is executed, no operation is performed. Execution proceeds on the statement following the REPEAT statement.

It is possible to branch into the range of a REPEAT...UNTIL loop, although this is poor programming practice.

Examples:

```
0005 X=1
0010 REPEAT
   :   X=X+X
   :   PRINT X
   : UNTIL X>100
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

UNTIL
BREAK
LOOP
REPEAT/UNTIL - Section 4.11.3 of the NPL Programmer’s Guide

RESAVE

General Form:

```
RESAVE [<[W] [S] [R]>] T [$][file-number,  ][!]prog-name
                              [disk-address, ]
                              [<address-var>,]
                              [line-number1][,[line-number2]]
```

Where:

```
prog-name    = an alpha-variable or literal containing the name
               of the program to be saved.

line-number1 = the lowest line-number of the program to be saved.

line-number2 = the highest line-number of the program to be
               saved.
```

Discussion:

The RESAVE command is used to save a program or a portion of a program to an existing disk file specified by the prog-name parameter. The existing disk file may be scratched or not scratched and may be a program or a data file. As with SAVE, if there is insufficient space to save the program in the file specified, the RunTime attempts to relocate the file to the end of the diskimage.

Operation of RESAVE is otherwise identical to SAVE. Please refer to SAVE for further details on the other parameters.

Examples:

```
0010 RESAVE T"MYPROG"
0010 RESAVE T/D11,Q$1000,2000
0010 RESAVE T<A$>,!"MYPROG"100,200
0010 RESAVE <W> T#1,Q$100
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

Please refer to the Compatibility Issues section of SAVE for further details on compatibility.

RESAVE (cont.)

In NPL Revision 4.0, the RESAVE command acts upon the current LIST module.

NPL Revision 4.0 or greater requires enough free memory to make a copy of the largest program line being saved to SAVE a program. Applications that run with very little free memory or which have very large program lines may need to do a CLEAR N or CLEAR V to free up enough memory to perform the SAVE operation.

WARNING--It is possible that a program may be loaded that does not have sufficient memory to save itself.

References:

RESET

General Form:

```
RESET function
```

Discussion:

The RESET function is used to terminate program execution. RESET is a function of the HELP Processor.

The RESET function invokes Immediate Mode and allows for normal Immediate Mode functions. Normal program continuation is not allowed. For this reason, RESET should be a last effort to terminate program execution because it does not allow program continuation, and is normally reserved for terminating a program when a system I/O error (e.g., disk not ready) cannot be handled in any other way.

While the program is unresolved in memory, variables are not cleared and can still be examined in Immediate Mode.

The RESET function performs the following:

- Closes all devices currently opened by the program.
- Clears all FOR/NEXT loop and subroutine stack information, including all FUNCTION and PROCEDURE calls.
- Turns TRACE Mode off and deactivates STEP Mode.
- Clears the screen and displays "READY (NPL) PARTITION #".
- Invokes Immediate Mode.

RESET (cont.)

The RESET function can be suppressed by setting byte 12 of the $OPTIONS system variable to HEX(01). Refer to $OPTIONS system variable for details.

The RESET function is available only in the interpretive RunTime program.

Examples:

Compatibility Issues:

On the Wang 2200, an actual RESET key is available.

References:

$OPTIONS
HELP Processor - Chapter 11 of the Programmer's Guide

RESTORE

General Form:

```
RESTORE [LINE line-number [,numeric-expression]]
        [ numeric-expression                  ]
```

Where:

```
0 <= numeric-expression < 65536
```

Discussion:

The RESTORE statement is used to reset the internal pointer for the next DATA item that is used for a READ statement. When RESTORE is specified without parameters, the internal pointer is reset to the first data item on the first DATA statement in the program. If a numeric-expression is specified, the data pointer is reset to the item in the DATA statements that would have that number if all the items were sequentially numbered from 1. A value of 0 is also legal and is equivalent to 1 in this content.

Each module has its own separate list of DATA items. However, all modules share a single global data pointer.

The data pointer is set by default to the start of a module when resolution of that module completes. A RUN statement sets the data pointer to the start of the RUN module's DATA list. Data items in other modules may not be read, until a RESTORE statement is executed in the other module.

A RESTORE statement always set the data pointer to a location in the current module's DATA list.

There is currently no mechanism to save or restore the current location of the global DATA pointer. Consequently, if a PUBLIC function, procedure or DEFFN' uses RESTORE or READ statements, it should be clearly documented that the DATA pointer is affected. Otherwise, this could constitute an unexpected side-effect to the logic of the calling program.

RESTORE (cont.)

When used with the LINE option, the data pointer is reset to the first DATA statement on the specified line number. If the specified line does not contain a DATA statement as the first statement on the line, an execution error occurs. If a numeric-expression follows the line number, the pointer is set to the item in the DATA statements that would have that number if all items starting at the specified LINE were sequentially numbered from 1. A value of 0 is also legal and is equivalent to 1 in this content.

The RESTORE pointer is reset to the start of the program by execution of the RUN statement or by any program overlay (LOAD statement).

Examples:

```
0010 RESTORE
0010 RESTORE 17
0010 RESTORE LINE 100
0010 RESTORE LINE 17,17*I
:0010 DIM A$16,B$16,C$16
:0020 FOR I=1 TO 3
    :    READ A$,B$,C$
    :    PRINT A$;B$;C$
    : NEXT I
:0030 PRINT
:0040 FOR I=1 TO 3
    :    RESTORE LINE 70,2
    :    READ A$,B$,C$
    :    PRINT A$;B$;C$
    : NEXT I
:0050 DATA "Niakwa"," NPL"," RunTime"
:0060 DATA "Program"," Release II"," Interpreter"
:0070 DATA "This"," is"," a"," test"
:RUN
Niakwa NPL RunTime
Program Release II Interpreter
This is a

is a test
is a test
is a test
```

Compatibility Issues:

In Wang 2200 Basic-2, if a line number specified in a restore statement does not have a DATA statement in it, the next line with a data statement is used.

References:

DATA
READ

RETURN

General Form:

```
RETURN
RETURN       (numeric-expression               )
RETURN       (alpha-variable or literal-string  )
RETURN ERROR (error-code                       )
```

Discussion:

The RETURN statement followed by either a numeric or alpha expression in parentheses indicates the end of execution of a FUNCTION. It does not mark the syntactic end of a function; this is provided by the END FUNCTION statement. A RETURN statement with a result-value is only legal within a FUNCTION body.

The RETURN ERROR statement followed by a numeric expression in parentheses indicates that the specified error condition should be raised in the calling statement. If the calling statement does not trap errors, the program is halted. The error-code should be a NPL error condition or user-defined error.

NOTE: Not all errors are recoverable, so if the error must be recoverable some error values should not be used. A specific range of error codes is reserved for user-defined or external error codes, both recoverable and non-recoverable. Refer to the ERROR statement for a definition of these ranges.

A RETURN ERROR statement is only legal within a FUNCTION or PROCEDURE body.

A RETURN statement with no options indicates the end of execution of a subroutine called using GOSUB or GOSUB'. An END PROCEDURE statement normally ends the execution of a PROCEDURE. If there are no pending GOSUB or GOSUB' calls since the last call to a function or procedure, the RETURN statement with no options may also be used to end the execution of a PROCEDURE (but not a FUNCTION).

Execution proceeds on the statement following the statement that called the subroutine or PROCEDURE.

RETURN (cont.)

When a RETURN statement ends the execution of a GOSUB or GOSUB', all incomplete FOR/BEGIN...NEXT loops which have been started since the subroutine was called are terminated, and the internal stack information for any such loops is cleared.

When any type of RETURN ends the execution of a FUNCTION or PROCEDURE, all incomplete FOR/BEGIN...NEXT loops and GOSUB or GOSUB' calls which have been started since the function or procedure was called are terminated, and the internal stack information for any such loops and subroutine calls is cleared.

When used with a subroutine entered through keyboard action (GOSUB'), the RETURN statement causes program control to be returned to:

1. Immediate Mode if the Specific Function key was pressed in Immediate Mode;
2. The INPUT statement if the Special Function key was pressed in response to an INPUT - input is reexecuted;
3. The statement following a LINPUT statement if the Special Function key was pressed in response to a LINPUT.

If a RETURN statement is encountered when a corresponding a GOSUB statement has not been executed, an error is generated (ERR P41 - RETURN Without GOSUB).

Examples:

```
0010 RETURN
0010 RETURN(result)

0010 RETURN(12)
0010 RETURN(FNX(A))
0010 RETURN(Temp$)
0010 RETURN("Wendy")
0010 RETURN ERROR(48)
```

RETURN (cont.)

Compatibility Issues:

References:

FUNCTION
PROCEDURE
GOSUB
GOSUB’

RETURN CLEAR

General Form:

```
RETURN CLEAR [ALL]
```

Discussion:

The RETURN CLEAR statement may be used to clear internal stack information created by the most recent GOSUB or GOSUB' subroutine call, without returning to the statement that made the subroutine call. Any incomplete FOR/BEGIN...NEXT loops which have been started since the subroutine was called are also terminated, and the internal stack information for any such loops is also cleared. Execution proceeds following the RETURN CLEAR statement.

Stack information created by GOSUB or GOSUB' statements prior to a pending call to a FUNCTION or PROCEDURE may not be cleared by RETURN CLEAR.

If a RETURN CLEAR statement is executed when there is no pending GOSUB or GOSUB' information, an error is generated (P41 - RETURN without GOSUB).

The RETURN CLEAR ALL statement clears all internal stack information for GOSUB and GOSUB' subroutine calls, as well as for any incomplete FOR/BEGIN...NEXT loops. Stack information created by GOSUB or GOSUB' statements or FOR/BEGIN...NEXT loops prior to a pending call to a FUNCTION or PROCEDURE is not cleared by RETURN CLEAR ALL.

WARNING--It is inadvisable to use RETURN CLEAR ALL when this could clear stack information for a PUBLIC DEFFN' subroutine, since this means that, in general, control cannot be transferred back to the calling program.

Structured programs should not require the use of RETURN CLEAR [ALL].

RETURN CLEAR (cont.)

Examples:

```
0010 RETURN CLEAR
0010 RETURN CLEAR ALL
:0010 GOSUB 100
:0020 PRINT "DONE": GOTO 400
:0100 GOSUB 200
   : PRINT "Returned from Subroutine 200"
   : RETURN
:0200 REM Subroutine line 200
   : GOSUB 300
   : PRINT "Returned from Subroutine 300"
   : RETURN

:0300 REM Subroutine line 300
   : PRINT "Before RETURN CLEAR"
   : LIST STACK
   : RETURN CLEAR
   : PRINT "After RETURN CLEAR"
   : LIST STACK
   : RETURN
:0400 REM Done
:RUN
Before RETURN CLEAR
0010 GOSUB 100
0100 GOSUB 200
0200 : GOSUB 300
After RETURN CLEAR
0010 GOSUB 100
0100 : GOSUB 200
Returned from Subroutine 200
DONE
```

NOTE: Executing the RETURN CLEAR statement in the subroutine on line 300 means that statements following the GOSUB 300 on line 200 are never executed.

Compatibility Issues:

References:

$REV

General Form:

```
     alpha-receiver = $REV
```

Discussion:

The $REV system variable contains the full revision number of the RunTime program currently executing as a twelve byte displayable alpha-numeric value in the format:

```
A.BB.CC.DD.H
```

Where:

A	The major release level. This indicates the primary syntactical and functional level of the release.
BB	The major revision level. This indicates a secondary level of syntactical or functional revisions.
CC	The minor revision level. This indicates minor changes in syntax or functionality. This level is typically used to distinguish beta or other prerelease versions from a customer ship version of the product.
DD	The edit level. This is usually set by patches or other field level changes.
H	The hardware version. This is the same as byte 1 of $MACHINE.

$REV is intended to be used to examine the revision level of a particular RunTime program for support and diagnostic purposes.

The complete revision level returned by $REV is also displayed on the RunTime startup screen.

Examples:

```
0010 X$=$REV
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

$REV is not supported on the Wang 2200.

References:

RND Function

General Form:

```
RND (numeric-expression)
```

Discussion:

The RND function produces a pseudo-random number greater than or equal to zero and less than one based on a numeric-expression. If the numeric-expression is zero, the RND function returns the first value in the pseudo-random number list. If the numeric-expression is non-zero the next value in the pseudo-random number list is returned. This is valid wherever a numeric-expression is legal.

A non-zero value is normally used when various random values are required. The zero value, when used once at the start of a program, ensures that the same sequence of pseudo-random numbers is used during testing and debugging of programs which use RND().

Examples:

```
0010 A = RND(1)
0010 G4 = INT(RND(1)*100)+1
0010 M2(K) = RND(0)+44
```

Compatibility Issues:

Due to the use of a different algorithm, results of this function may differ from functions evaluated on a Wang 2200. The first value of the pseudo-random chain (produced by RND(0)) is returned as .1584625767084 which is the same as on the Wang 2200 MVP (some programs use this value to decide whether they are running on a Wang 2200T processor) but subsequent values returned by RND(1) are, in general, different from those on the Wang 2200 MVP.

References:

ROTATE

General Form:

```
ROTATE [C] (alpha-variable,numeric-expression)
```

Where:

```
-8 <= numeric-expression <= 8
```

Discussion:

The ROTATE statement is used to rotate data at the bit level of the specified alpha-variable on a character-by-character basis. All bytes of the specified alpha-variable are ROTATEd including trailing spaces.

The numeric-expression specifies the number of bits to be rotated and can have values from -8 to 8, inclusive. A positive numeric-expression causes the bits to be rotated to the left. A negative numeric-expression causes the bits to be rotated to the right.

If the "C" option is used, the entire alpha-variable is treated as a single string of bits during the rotate operation. If the "C" option is not used, the rotate occurs only within each byte of the alpha-variable.

ROTATE C can be used to rotate characters within an alpha-variable by specifying a numeric-expression of 8 or -8.

ROTATE is useful in multi-precision or BCD arithmetic.

Examples:

```
0010 ROTATE(X$,1)
0010 ROTATE(Q$,-3)
0010 ROTATE(STR(Q$,6,4),4)
0010 ROTATE C(X$,A)
0010 ROTATE C(STR(X$,3,7),-8)
```

ROTATE (cont.)

```
:0010 DIM A$4
:0020 A$=HEX(B4829410)
:0030 ROTATE(A$,4)
:0040 PRINT "AFTER ROTATE  A$= HEX ";
:0050 HEXPRINT A$
:0060 A$=HEX(B4829410)
:0070 ROTATEC(A$,4)
:0080 PRINT "AFTER ROTATEC A$= HEX ";
:0090 HEXPRINT A$

:RUN
AFTER ROTATE  A$= HEX 4B284901

AFTER ROTATEC A$= HEX 4829410B
```

Compatibility Issues:

References:

ROUND Function

General Form:

```
     ROUND(round-value,round-factor)
```

Where:

```
     round-value  = a numeric-expression whose value is to be
                    rounded.

     round-factor = a numeric-expression specifying the rounding fac-
                    tor. Valid wherever a numeric-expression is le-
                    gal.
```

Discussion:

The ROUND function is used to round a numeric-expression to a specified decimal place.

If the round-factor is greater than zero, the round-value is rounded to the number of digits beyond the decimal point indicated by the round-factor. If the round-factor is less than zero, the round-value is rounded to that position to the left of the decimal.

Examples:

```
0010 A=ROUND(B,C)
0010 A=ROUND(B,2)
0010 A=ROUND(351,2)

:PRINT ROUND(1.2579,1)
 1.3
:PRINT ROUND(1.2579,3)
 1.258
:PRINT ROUND(671,-2)
 700
:PRINT ROUND(671,-1)
 670
```

Compatibility Issues:

References:

RUN Command

General Form:

```
RUN [line-number [,statement-number]]
```

Discussion:

The RUN command is used to initiate execution of the program currently in memory.

Before execution of the program begins, the program is first resolved in memory. During program resolution, a check is performed for syntax errors, variable and line-number references. If errors are encountered, an error message is displayed and execution is terminated.

When the RUN command is executed with no line-number specified, non-common variables are cleared from memory, GOSUB and FOR/TO information is cleared from the internal stack, program resolution occurs, and program execution begins.

Specifying a line-number causes the program to be resolved in memory and program execution begins at the specified line. All variables (common and non-common) remain undisturbed in memory. If the specified line-number does not exist in the program, a runtime error P36 (Undefined Line Number or Continue Illegal) is generated.

The statement-number parameter allows execution to begin on a particular statement within a multi-statement program line. Statements are numbered within a program line from left to right, starting at 1.

Execution of the RUN command as a program statement is allowed. However, the line-number and statement-number parameters are not allowed when used as a program statement. Refer to RUN statement for details.

RUN Command (cont.)

Examples:

```
:RUN
:RUN 100
:RUN 10,2
:0005 Y$="9/30/86"
:0010 PRINT "Today’s date is: ";Y$
:0020 PRINT "Company Name: "        : REM (statement-number 1)
    : X$="Niakwa"                   : REM (statement-number 2)
    : PRINT X$                      : REM (statement-number 3)
:RUN
Today’s date is: 9/30/86
Company Name: Niakwa

:RUN 20,2
Niakwa
```

Execution of the RUN 20,2 command causes execution to begin at line 20, statement-number 2 (X$="Niakwa").

Compatibility Issues:

This statement is supported only with Release 2.0 or greater.

Execution of the RUN command in NPL Revision 4.0 resets the current module to the root (no name) module.

References:

LOAD RUN
RUN Statement

RUN Statement

General Form:

```
     RUN module-name[,return-var]
```

Where:

```
     module-name = { alpha-variable   }
                   { literal-string   }

     return-var  = { numeric-scalar   }
                   { num-array-element}
```

Discussion:

The RUN statement is used to reinitiate execution of the program currently in memory.

Execution of the RUN statement has the following effects:

- The LIST module is reset to the RUN module.
- All FUNCTION and PROCEDURE return information is cleared from the stack.
- All GOSUB/RETURN and FOR/NEXT loop information is cleared from the stack.
- All non-common variables are cleared (CLEAR N).
- The program is reresolved and execution starts from the beginning.

The programmable RUN statement allows a program to perform dynamic dimensioning of variables without reloading itself from disk. However, Revision 4.00 or greater allow dynamic dimensioning of arrays using MAT REDIM.

RUN Statement (cont.)

RUN as a program statement could also be used in an application that should completely "restart" (including clearing all non-common variables and resetting them to blanks or zeroes).

Examples:

```
0010 RUN
0010 IF S>1 THEN RUN
0010 RUN PlotDriver$
0010 RUN "REPORT"
0010 RUN "AccountsReceivableSystemMenu",ExitCode

:0010 COM S
   :DIM A$(S)1       :REM Want A$() array as large as possible
    :IF S>0 THEN 20  :REM first time, S=0
    :S=SPACE-100     :REM after other variables are resolved
    :RUN             :REM use rest of memory then re-run
:0020 REM start program
    :LIST DIM
:RUN
DIM A$(57222)1
COM S
```

Compatibility Issues:

This statement is supported only with Release 2.0 or greater.

Wang 2200 Basic-2 does not allow RUN as a programmable statement.

References:

SAVE

General Form:

```
SAVE [<[W][S][R]>] T [$]  [file-number,    ][([old-prog-name ])][!]
                         [disk-address,   ][space          ]
                         [<address-var>,  ]
          new-prog-name  [line-number1    ][,[line-number2] ]
```

Where:

```
space         = a numeric-expression which represents the number
                of extra sectors to reserve in addition to the
                program.

old-prog-name = an alpha-variable or literal containing the name
                of an existing scratched file to be overwritten.

new-prog-name = an alpha-variable or literal containing the name
                of the program to be saved.

line-number1  = the lowest line-number of the program to be saved.

line-number2  = the highest line-number of the program to be
                saved.
```

Discussion:

The SAVE statement is used to save a program or a portion of a program currently in memory to a designated diskimage. The disk catalog index is updated with the program name, start-sector address, and end-sector address saved. All program lines from line-number1 through line-number2 are saved on disk. If line-number1 is not specified, all program lines from the first line in memory to line-number2 is saved. If line-number2 is not specified, all program lines from line-number1 to the last line in memory are saved. If no line-numbers are specified, the entire program in memory is saved.

NOTE: Presence of the "$" parameter specifies that a read-after-write is to be executed as program text is saved on disk to verify that all text is written correctly to the disk.

SAVE (cont.)

WARNING--If neither old-prog-name nor space parameters are specified, the file is saved as a new file with space defaulting to a value of 0. If empty parenthesis () are specified, the old-prog-name defaults to the same name as new-prog-name (e.g., SAVE overwrites the scratched file with the same name as new-prog-name). If the scratched file does not exist with the same name as new-prog-name, an error is generated (D84 - File Not Scratched).

NOTE: The "!" parameter specifies that the program is to be saved in scramble protected format. The entire program is scrambled as it is saved to the disk. This discourages examination or modification of the program.

The SAVE statement cannot be used if any part of the currently loaded program was scramble protected at the time it was loaded.

When issued from Immediate Mode, the SAVE command relocates files at the end of the diskimage if the new-prog-name is the same as the old-prog-name (previously scratched file) and insufficient space is available in old-prog-name. Otherwise, if insufficient space is available in the diskimage, an error is generated.

On the non-interpretive RunTime program, executing SAVE generates an error unless the line-number range does not include **any** program text.

Examples:

```
SAVE T "SP LOAD"
SAVE T "SP MENU"8000
SAVE T !"SECURITY"
SAVE T#2,"START"8000,8100
SAVE T !Q$
SAVE T#Q,()"PROGRAM"100,2000
SAVE T<A$>,()"PROGRAM"100,2000
SAVE T ("PROGRAM")"PROGRAM2"
SAVE T/D20,(10)X$
SAVE T ()X$
```

SAVE (cont.)

Compatibility Issues:

In Wang 2200 Basic-2, programs are saved in atomized 2200 executable format. In NPL, programs are saved in p-code format.

In Wang 2200 Basic-2, the "< S>" parameter specifies that all spaces within a program are to be removed as it is stored on disk. NPL supports the < S> parameter syntax for compatibility purposes, but this parameter has no effect on the SAVE statement at run-time.

In Wang 2200 Basic-2, the "< SR>" parameter specifies that all insignificant spaces and REM’s within a program are to be removed as it is stored on disk. NPL supports the < SR> parameter syntax for compatibility purposes, but this parameter has no effect on the SAVE statement at runtime.

In Wang 2200 Basic-2 Revision 3.0 and higher, the < W> parameter specifies that sector boundaries are to be ignored when saving the program. Sector boundaries are always ignored in NPL. Therefore this operation performs no operation in NPL. It is supported for syntactical compatibility only.

The "< W>" parameter is recognized only on NPL Revision 3.0 or greater.

Use of the address-var parameter is supported only on NPL Revision 3.0 or greater and is not supported on the Wang 2200.

In NPL, REM’s can be removed as they are entered by setting the system variable $KEEPREMS to HEX(00). Refer to $KEEPREMS system variable for details on removing REM’s and spaces within a program. REM’s can also be removed when compiling programs by specifying the compiler KEEPREMS option OFF. Refer to Compiler Operations, Chapter 14 of the Programmer’s Guide for details.

In Wang 2200 Basic-2, a "P" parameter may be specified in place of the "!" parameter which specifies that the program is to be "Protected" but not scrambled. This is not supported by NPL.

In Wang 2200 Basic-2, programs are not automatically relocated at the end of the diskimage.

In NPL Revision 4.0, the SAVE command acts upon the current list module.

SAVE (cont.)

NPL Revision 4.0 or greater requires enough free memory to make a copy of the largest program line being saved to SAVE a program. Applications that run with very little free memory or which have very large program lines may need to do a CLEAR N or CLEAR V to free up enough memory to perform the SAVE operation.

WARNING--It is possible that a program might be loaded with insufficient memory to save it.

References:

$KEEPREMS
OBJFORMAT Option - Section 14.7 of the Programmer's Guide
Loading Programs - Section 5.3 of the Programmer's Guide

SAVE BOOT Command

General Form:

```
     SAVE BOOT [progname]
```

Where:

```
     progname = an alpha-variable or literal-string containing the na-
                tive file-specification of a bootstrap program.
```

Discussion:

The SAVE BOOT command is used to save bootstrap programs to the native file system. A bootstrap program is a NPL program which is saved as a native file and is automatically loaded and executed by the RunTime at initial start up. Refer to the appropriate NPL Supplement for details.

When progname is omitted or blank, the "default" boot program name is assumed. Initially, the "default" boot program name is either BOOT, or the name of the boot program specified on the command line when the RunTime Program was invoked.

If the native operating system allows extensions, a .OBJ extension is assumed or no extension is specified.

The "default" boot program name is changed any time a LOAD BOOT or SAVE BOOT command is entered with an explicit filename.

When saving a bootstrap program, any existing file in the current directory with the same name is replaced.

For immediate mode SAVE BOOT commands, program text from the current LIST module is saved. For other SAVE BOOT commands, program text from the executing module is saved.

NOTE: Scramble protection and partial program saving (line-number ranges) options are not supported by the SAVE BOOT command.

SAVE BOOT cannot be used if any part of a currently loaded program was scramble-protected at the time it was loaded.

SAVE BOOT (cont.)

Programs saved by SAVE BOOT may also be used by the PREBOOT (/P) option. Refer to Chapter 4 of the appropriate Supplement for information on the preboot option.

Examples:

```
:SAVE BOOT
:SAVE BOOT "UTILITY"   :REM       Saves a program named "UTILITY.OBJ"
                                  to the currently selected native
                                  file system directory.
```

Compatibility Issues:

This statement is supported only with Release 2.0 or greater.

The SAVE BOOT command is not a valid instruction in Wang 2200 Basic-2.

In NPL Revision 4.0, the SAVE command acts upon the current list module.

References:

LOAD BOOT
Loading Programs - Section 5.3 of the Programmer's Guide

SAVE DA

General Form:

```
     SAVE DA [<[W][S][R]>] T[$] [file-number,   ](expr1[,return-value])
                                [disk-address,   ]
                                [<address-var> , ]
                                [line-number1   ][,[line-number2]]
```

Where:

```
     expr1        = an alpha-variable or numeric-expression.

     return-value = an alpha-variable or numeric-expression.

     line-number1 = the lowest line-number in the program to be saved.

     line-number2 = the highest line-number in the program to be saved.
```

NOTE: The use of this statement is not recommended. Refer to SAVE as a better alternative.

Discussion:

The SAVE DA command is used to save a program or a portion of a program currently in memory into a designated diskimage without accessing the catalog index. All program lines from line-number1 to line-number2 are saved on disk. If line-number1 is not specified, all program lines from the first line-number in memory to line-number2 is saved. If line-number2 is not specified, all program lines from line-number1 to the last line-number in memory is saved.

NOTE: Presence of the "$" parameter specifies that a read-after-write is to be executed as program text is saved on disk to verify that all text was written correctly.

Expr1 contains the first sector-number to be saved. If expr1 is an alpha-variable, the binary value of the first two bytes is used.

Use of an alpha-variable to contain sector addresses results in improper sectors being accessed if extended (greater than 16 MB) diskimages are in use and the sector numbers being accessed are greater than 65355. Refer to Section 7.3.10 of the Programmer's Guide for further programming considerations for use of extended diskimages.

SAVE DA (cont.)

The return-value performs no operation in NPL. The return-value does not affect operation of the SAVE DA statement at runtime. No value is returned to the return-value if specified.

On the non-interpretive RunTime Program, executing SAVE generates an error unless the line-number range does not include any program text.

No catalog information is read by SAVE DA. It is the programmer's responsibility to ensure that the program is saved at the correct sector address and does not overwrite other programs or data files unintentionally. Use of SAVE DA is to be avoided where possible in favor of cataloged SAVE statements.

Examples:

```
SAVE DAT(100)
SAVE DAT$/D12,(Q,Q)8000,8100
SAVE DAT#2,(Q$,Q$)
SAVE DAT<A$>,(Q$,Q$)
SAVE DAT#Q,(1000)
```

Compatibility Issues:

In Wang 2200 Basic-2, programs are saved in atomized 2200 executable format. In NPL, programs are saved in p-code format.

In Wang 2200 Basic-2, the "< S>" parameter specifies that all spaces within a program are to be removed as it is stored on disk. NPL supports the < S> parameter syntax for compatibility purposes, but this parameter has no effect on the SAVE statement at runtime. In Wang 2200 Basic-2, the "< SR>" parameter specifies that all insignificant spaces and REM's within a program are to be removed as it is stored on disk. NPL supports the < SR> parameter syntax for compatibility purposes, but this parameter has no effect on the SAVE statement at runtime.

In Wang 2200 Basic-2 Revision 3.0 and higher, the "< W>" parameter specifies that sector boundaries are to be ignored when saving the program. Sector boundaries are always ignored in NPL. Therefore this option performs no operation in NPL. It is supported for syntactical compatibility only.

SAVE DA (cont.)

The "< W>" parameter is recognized only on NPL Revision 3.0 or greater.

Use of the address-var parameter is supported only on NPL Revision 3.0 or greater and is not supported on the Wang 2200.

In NPL, REM’s can be removed as they are entered by setting the system variable $KEEPREMS to HEX(00). Refer to $KEEPREMS system variable for details on removing REM’s and spaces within a program. REM’s can also be removed when compiling programs by specifying the compiler KEEPREMS option OFF. Refer to Compiler Operations, Chapter 14 of the Programmer’s Guide for further details.

In Wang 2200 Basic-2, the return-value returns the sector immediately following the last sector accessed by the SAVE DA operation. The return-value does not affect operation of the SAVE DA statement in NPL. No value is returned in the return-value if specified. The syntax is supported for compatibility purposes only.

In NPL Revision 4.0, the SAVE DA command acts upon the current list module.

References:

$KEEPREMS
OBJFORMAT Option - Section 14.7 of the Programmer’s Guide
Loading Programs - Section 5.3 of the Programmer’s Guide

SCRATCH

General Form:

```
     SCRATCH T [file-number,  ] file-name [,file-name]...
               [disk-address, ]
               [<address-var>,]
```

Where:

```
     file-name = an alpha-variable or literal-string containing the
                 name of the existing cataloged file to be scratched.
```

Discussion:

The SCRATCH statement is used to set the status of a file or files to a scratched condition. It does not delete the filename from the catalog index nor does it alter the contents of the file itself except for the file trailer sector. The sectors used by scratched files can be reused to save new programs or data files.

A file is scratched to allow its contents to be overwritten (using DATA SAVE DC OPEN or SAVE statements) or to remove its name from the catalog index and its contents from the catalog area by execution of a MOVE statement.

NOTE: **As of Revision 3.0 of NPL, programs may be resaved without first scratching them by use of the RESAVE statement.**

Once a file has been scratched, it is no longer accessible with DATALOAD DC OPEN or LOAD statements, although it can be renamed or reused by DATASAVE DC OPEN or SAVE statements.

NOTE: **As of Revision 3.0 of NPL, files can be renamed without first scratching them by use of the RENAME statement.**

As of Revision 3.0 of NPL, files which are SCRATCHed in error may be set back to normal status by use of the UNSCRATCH statement.

SCRATCH (cont.)

Examples:

```
0010 SCRATCH T"START",Q$
0010 SCRATCH T X$,X1$,X2$
0010 SCRATCH T#Y,"SP MENU"
:SCRATCH T"START"
:SCRATCH T/D32,"START","SP START","SECURITY"
:SCRATCH T<A$>,"PROGRAM"
```

Compatibility Issues:

Use of the address-var parameter is supported only on NPL Revision 3.0 or greater and is not supported on the Wang 2200.

References:

SAVE
LOAD
DATA SAVE DC OPEN
DATA LOAD DC OPEN
MOVE
RESAVE
UNSCRATCH

SCRATCH DISK

General Form:

```
SCRATCH DISK [’]T [file-number,  ][LS=value1,] END=value2
                  [disk-address, ]
                  [<address-var>,]
```

Where:

```
’      = specifies the alternate hashing method is to be used.

value1 = a numeric- expression which represents the required size
         of the Catalog Index whose value is from 1 to 255.  The
         default value if not specified is 24.

value2 = a numeric- expression which represents the highest sec-
         tor address in the Catalog area.  The expression must be
         less than or equal to highest sector address available
         on disk.
```

Discussion:

The SCRATCH DISK statement is used to create a new diskimage or to delete and recreate existing diskimages.

The SCRATCH DISK statement is also used to define the physical size of a disk device. When directed to diskimage files, the diskimage is deleted (if already existing) and then recreated to the size of the specified END= expression, provided physical disk space is available. When directed to "raw" diskette devices, the index is cleared and the physical end of catalog is set.

The ’ option specifies the alternate hashing method is to be used when scratching the diskimage.

The "LS" parameter is used to specify the size of the catalog index. If no "LS" parameter is specified, a default value of 24 is used. The first index sector in the catalog index can hold up to 15 index entries, all other sectors in the catalog index hold up to 16 index entries. A maximum of 255 sectors is allowed for the catalog index size. The formula for calculating the maximum number of files in an index catalog is:

SCRATCH DISK (cont.)

(assuming LS=X)
Maximum # of Files = (X-1)*16+15

The "END" parameter is used to specify the end of the catalog area to be used for storing cataloged files.

WARNING--This statement removes all data and programs from an existing diskimage and should be used with appropriate caution.

Examples:

```
0010 SCRATCH DISK T LS=127, END=19583
0010 SCRATCH DISK T#1, END=3873
0010 SCRATCH DISK T END=32607
0010 SCRATCH DISK T/D11, LS=11, END=1231
0010 SCRATCH DISK T<A$>, LS=11, END=1231
0010 SCRATCH DISK T LS=Q, END=Q1
0010 SCRATCH DISK T#X, LS=31, END=X2
0010 SCRATCH DISK 'T/D12, LS=20, END=4799

:SCRATCH DISK T/D60,LS=10,END=1279
:LIST DC T/D60
$DEVICE(/D60) ="/BASIC2C/PROGS.BS2"
INDEX SECTORS =     10
END SECTORS   =   1279
CURRENT END   =      9

FILE    TYPE START   END  USED  FREE    DATE     TIME
:
```

Compatibility Issues:

In NPL, the SCRATCH DISK statement performs the function of defining the size of a diskimage file. The diskimage file is actually deleted and then recreated to the size of the specified END= value. In the Wang 2200 Basic-2, devices are not actually deleted and recreated. The physical device is simply cleared to the specified size.

Operation of SCRATCH DISK is operating system dependent. Refer to Diskimage Files and Raw Disk Devices in the NPL Supplement(s) for details.

Use of the address-var parameter is supported only on NPL Revision 3.0 or greater and is not supported on the Wang 2200.

SCRATCH DISK (cont.)

References:

Native Operating System Files as Diskimages - Section 7.3.4 of the Programmer's Guide
Native Operating System "Raw" Devices as Diskimages - Section 7.3.5 of the Programmer's Guide

$SCREEN

General Form:

Form 1:

`$SCREEN=`*`alpha-expression`*

Form 2:

`alpha-receiver``=$SCREEN`

Where:

`alpha-expression` `= length of 256 characters.`

Discussion:

This statement allows a NPL application program to examine or modify the current screen translation table. Form 1 allows the $SCREEN system variable to be modified. Form 2 allows the $SCREEN system variable to be examined.

The $SCREEN system variable contains the 256-byte screen translation table currently in effect. The screen translation table contains the character equivalents to be sent to the screen in place of the character received from the NPL program. Byte 1 contains the replacement character for HEX(00), byte 256 contains the replacement character for HEX(FF), etc.

For example:

```
10 DIM X$(256)1
20 X$()=$SCREEN        : REM PLACE CURRENT TABLE IN X$
30 PRINT STR(X$(),66,2): REM RESULTS WOULD BE "AB"
40 STR(X$(),66,1)="B"  : REM REPLACE CHARACTER IN POSITION 66
                             (VAL(HEX(41))+1) WITH THE CHARACTER "B"
50 $SCREEN=X$()        : REM MODIFY SCREEN TRANSLATION TABLE
```

The effect of this character is that whenever the character "A" is sent to the screen, the character "B" appears.

$SCREEN (cont.)

Changes made to the screen translation table using the $SCREEN statement go into effect immediately and remain in effect, unless further modifications are made or until the end of the current RunTime session. Changes are not retained from one session to the next. To make more permanent modifications to the screen translation table, please use the Screen Translation Table Editor Utility. Refer to the appropriate NPL Supplement for further details.

NOTE: The values of characters in the $SCREEN system variable refer to the character set of the native operating system. Available character sets vary from one machine to another. Refer to the appropriate NPL Supplement for details.

Compatibility Issues:

$SCREEN statement is not a valid instruction in Wang 2200 Basic-2.

Default values for $SCREEN vary from one machine to another. Refer to the appropriate NPL Supplement for hardware-specific details.

References:

Screen Translation Table Editor - Chapter 13 of the Programmer's Guide
Screen Handling - Chapter 7 of the Programmer's Guide

SELECT

General Form:

```
SELECT select-item [,select-item]...
```

Where:

```
select-item           = { PRINT    } select-specification
                        { LIST     }
                        { CO       }
                        { CI       }
                        { INPUT    }
                        { DISK     }
                        { TAPE     }
                        { PLOT     }
                        { D,R,G    }
                        { ERROR    }
                        { ROUND    }
                        { P        }
                        { LINE     }
                        { @PART    }
                        { #expr    }
                        { LOG      }
                        { TERMINAL }
                        { ON,OFF   }
                        { DRIVER   }
                        { ON CLEAR }
                        { TC       }
                        { LISTLINE }

expr                  = an integer, numeric-scalar or numeric-ar-
                        ray-element whose value is a valid device
                        number
                        (0 <= expr <= 255).

select-specification  = defined by individual SELECT statements.
```

SELECT (cont.)

Discussion:

The SELECT statement is used to assign multiple select-items separated by commas. Refer to the individual SELECT statements for detailed information on each select-item.

Examples:

Compatibility Issues:

References:

$SELECT

General Form:

```
     alpha-receiver = [$]SELECT{(internal-device-table-entry)}
                               {internal-device-table-entry  }
```

Where:

```
     internal-device-table-entry  = { PRINT [WIDTH]}
                                    { LIST [WIDTH] }
                                    { CO [WIDTH]   }
                                    { CI    }
                                    { INPUT }
                                    { DISK  }
                                    { TAPE  }
                                    { LOG   }
                                    { PLOT  }
                                    { #expr }

     expr                         = a numeric-expression whose value
                                    is a valid device number
                                    (0 <= expr <= 255).
```

Discussion:

The $SELECT function allows program inspection of the currently established default devices in the Internal Device Table. The returned value is a 3-byte device-address in ASCII format for all device specifications except PRINT, LIST, and CO when the optional WIDTH keyword is specified. For PRINT, LIST, and CO when WIDTH is specified, the returned value includes the 3 byte device-address followed by a WIDTH field. The WIDTH field may be 3 to 5 characters in length and is comprised of the current width value in ASCII enclosed by parenthesis. There are no spaces between the device-address and the WIDTH field returned.

The $SELECT function could be used to preserve the status of, for example, the SELECT PRINT address. Selecting the screen address, operator messages can then be directed to the screen before reselecting the original PRINT address.

$SELECT (cont.)

NOTE: Since the width parameter, if specified, is returned as part of an alpha-variable, it can not be used directly to reestablish the width when reselecting a device. It must be converted to a numeric value first.

For example:

```
0010 A$=SELECT PRINT WIDTH
0020 SELECT PRINT 005(80)
0030 PRINT "Message to the operator"
0040 CONVERT STR(A$,5,NUM(STR(A$,5))) TO W: REM extract width
0050 SELECT PRINT <A$>(W): REM reselect original address and width
```

In Immediate Mode, the values of the Internal Device Table are more easily inspected using LIST DT. However, if the current value of the SELECT LIST device is unknown, it can best be determined by use of $SELECT.

Examples:

```
0010 X$=SELECT PRINT WIDTH
0010 Y$=SELECT DISK
0010 X$=SELECT LIST

:0010 SELECT #1/D20
:0020 A$=SELECT #1
:0030 PRINT "ADDRESS #1=";A$
:0040 A$=SELECT PRINT WIDTH
:0050 PRINT "CURRENT PRINT ADDRESS =";A$
:RUN
ADDRESS #1=D20
CURRENT PRINT ADDRESS=005(80)
```

Compatibility Issues:

The $SELECT function is implemented in Revision 2.00 and greater of NPL.

As of Revision 3.0 and higher of NPL, the syntax of $SELECT has been modified to support the Wang Basic-2 syntax in addition to the original NPL syntax. The $ preceding the statement and the parenthesis around the specified internal-device-table-entry are now optional. Previously supported syntax is still recognized by the compiler and RTI. Which syntax is generated by the de-compiler is determined by the status of the $KEEPREMS (in RTI) or the KEEPREMS option (in B2C) when the statement is originally compiled. Statements compiled with KEEPREMS ON using Revision 3.0 or greater of B2C or RTI are de-compiled using the same syntax as entered. Statements compiled with KEEPREMS OFF or with earlier versions of B2C or RTI are de-compiled using the original NPL syntax.

$SELECT (cont.)

The Width parameter for PRINT, LIST, and CO is supported only in Revision 3.0 or higher of NPL. The Width parameter is not supported in Wang Basic-2.

The LOG internal device table entry is supported only in NPL Revision 3.0 or greater and is not supported on the Wang 2200.

References:

SELECT
LIST DT
Internal Device Table - Section 7.2.3 of the Programmer's Guide

SELECT @PART

General Form:

```
     SELECT @PART {alpha-variable}
                  {literal-string}
```

NOTE: This statement is supported for Wang compatibility reasons only and its use in new development performs no operation.

Discussion:

The syntax of this statement is supported only for compatibility with Wang 2200 Basic-2. When executed under NPL, the alpha-variable or literal-string must be blank or an error X77-Invalid Partition Reference is generated.

Examples:

Compatibility Issues:

In Wang 2200 Basic-2, the SELECT @PART instruction specifies that a global partition is to be referenced by the partition SELECT @PART is executed from. In NPL, global partitions are not supported. This statement is supported for syntactical purposes only.

Global partitions are not supported by NPL.

References:

SELECT CI

General Form:

```
     SELECT CI {device-address  }
               {<alpha-variable>}
```

Where:

```
     alpha-variable = an alpha-variable in which the first three bytes
                      contain an ASCII representation of a hexdigit
                      (0-9; A-F).
```

NOTE: This statement is supported for Wang compatibility reasons only and its use in new development performs no operation.

Discussion:

The SELECT CI statement sets the Console Input entry in the Internal Device Table to the specified device-address. This entry in the Internal Device Table performs no operation. This statement is supported for syntactical purposes only.

The slash designation used as part of the standard device-address is optional with this statement.

Examples:

```
0010 SELECT CI 001
:SELECT CI 001
```

Compatibility Issues:

In Wang 2200 Basic-2, the Console Input entry in the Internal Device Table specifies the device-address for keyboard input. In NPL, this entry in the Internal Device Table performs no operation (NOP). Keyboard input is always entered from the default device-address /001. This statement is supported for syntactical purposes only.

References:

SELECT CO

General Form:

```
     SELECT CO {device-address  } [(width)]
               {<alpha-variable>}
```

Where:

```
     alpha-variable = an alpha-variable in which the first three bytes
                      contain an ASCII representation of a hexdigit
                      (0-9; A-F).

     width          = a numeric-expression indicating optional line
                      width for the output device, in the range 0-255.
```

NOTE: **Discussion:**

The SELECT CO statement sets the Console Output entry in the Internal Device Table to the specified device-address. All output produced by TRACE Mode is directed to the specified Console Output device-address (usually the screen or printer).

The width parameter is used to format the CO output to a designated line width. If the width is omitted, the lastline width selected for a CO operation is used. The default line width established at boot time is set equal to the maximum line width of the primary terminal. A specified line width of (0) indicates that line width is disregarded entirely, and that a carriage return is not issued until the entire output line has been printed, regardless of length.

Programmers should be aware that, in addition, a CLEAR command (with no parameters specified) copies the CO address and width to the SELECT PRINT and LIST addresses and widths.

The slash designation used as part of the standard device-address is optional with this statement.

SELECT CO (cont.)

Examples:

```
:SELECT CO 215
:SELECT CO 005
0010 SELECT CO 215
```

Compatibility Issues:

References:

TRACE

SELECT D,R,G

General Form:

```
SELECT {D}
       {R}
       {G}
```

Where:

```
D = specifies Degrees.

R = specifies Radians.

G = specifies Gradians.
```

Discussion:

The SELECT {D,R,G} statement sets the mathematics mode entry in the Internal Device Table to the specified type. The default mathematical mode entry in the Internal Device Table is Radians. This entry can be changed by executing another SELECT {D,R,G} statement or by reentering the RunTime Program.

The mathematical mode entry specifies the type of measure being used during trigonometric functions (SIN, COS, TAN, ARCSIN, ATN, ARCCOS).

The value of 360 degrees is represented as follows using the different units of measure:

- 2*#PI radians
- 400 gradians
- 1 revolution

SELECT D,R,G (cont.)

Examples:

```
0010 SELECT R
0010 SELECT G
0010 SELECT D
:0010 SELECT R
    : PRINT SIN(360)
:0020 SELECT D
    : PRINT SIN(360)
:0030 SELECT G
    : PRINT SIN(360)
:RUN
.95891572342024
 0
-.58778525229251
```

Compatibility Issues:

References:

SELECT DISK/FILE-NUMBER

General Form:

```
     SELECT {DISK    } {device-address  }
            {#expr   } {<alpha-variable>}
```

Where:

```
     alpha-variable = an alpha-variable in which the first three bytes
                      contain an ASCII representation of a hexdigit (0-
                      9; A-F).

     expr           = an integer, numeric-scalar or numeric-array-ele-
                      ment whose value is a valid device number.
```

Discussion:

The SELECT DISK statement sets the Current Disk entry in the Internal Device Table to the specified device-address.

This entry in the device table is used for any disk instructions which do not explicitly specify a disk-address or file-number (i.e., LIST DC, DATALOAD, SAVE, etc.).

The #expr specifies the file-number slot in the Internal Device Table to be assigned the specified device-address. Specifying file-number #0 is the equivalent of the DISK parameter.

Valid File #s

File #0 through file #15 are always valid file #s. Additional file #s may be defined by use of an expression which is a constant in the range of 16 to 255. For example:

```
SELECT #32 /D30
```

establishes file #0 through file #32 as valid.

For each additional file # defined, 24 bytes of the user partition are allocated. Should insufficient memory be available, an A01 error (Memory Overflow, Text < ---> Variable Table) occurs.

SELECT DISK/FILE-NUMBER (cont.)

Allocation of additional file #s takes place at resolution time. If multiple SELECT # constant statements are present, the highest constant specified becomes the highest valid file #. Once allocated, file #s above 15 are de-allocated only by the CLEAR and LOAD RUN statements.

Attempting to use a variable for a file # above 16 generates a P34 error (Illegal Value) if the file # specified exceeds the current defined maximum as described above.

For example:

```
0010 SELECT #32 /D30
0020 A=32
0030 SELECT #A /D30
0040 B=33
0050 SELECT #B /D30
```

produces a P34 (Illegal Value) error on line 50 since the value of B (33) is higher than the currently defined file # maximum (set at 32 in line 10).

Examples:

```
0010 SELECT DISK/D11
0010 SELECT #1/215,#2/D11,#3/D10
0010 SELECT #4/D12
0010 SELECT #A1<A$>
:SELECT DISK/D10
:SELECT DISK<A$>
:SELECT #1/D11
```

Compatibility Issues:

Use of file #s above #15 is supported only on NPL Revisions 3.0 and greater.

The method of implementation for support of file #s greater than #15 is fully compatible with the method used in Wang Basic-2 Revision 3.1.

References:

Internal Device Table - Section 7.2.3 of the Programmer’s Guide

SELECT DRIVER

General Form:

```
     SELECT DRIVER {device-address  } [OFF]
                   {<alpha-variable>}
```

Where:

```
     alpha-variable = an alpha-variable in which the first three
                      bytes contain an ASCII representation of a
                      hexdigit (0-9; A-F).
```

The compiler generates a warning when this statement is encountered.

Discussion:

The syntax of this statement is supported only for compatibility with Wang 2200 Basic-2. No operation is performed when this statement is encountered at execution time.

Examples:

Compatibility Issues:

This instruction, when executed on a Wang 2200 MVP, is used to enable or disable use of the generalized printer driver.

Under NPL, the generalized printer driver is not supported.

This statement is syntactically recognized only by NPL Revision 3.0 or greater.

References:

SELECT ERROR

General Form:

```
SELECT ERROR [ > error-code]
```

Where:

```
error-code = any computational error-code (60-69).
```

Discussion:

The SELECT ERROR statement is used to suppress system error messages caused by computational errors. Computational errors are those errors produced during the execution of arithmetic operations or functions.

NOTE: The value for SELECT ERROR affects all modules.

Normally, computational errors cause program execution to terminate followed by an error message. The SELECT ERROR statement can be used to suppress the error condition and allow the program to continue normally. The value of the receiver after a computational error varies depending on the type of error encountered. The computational errors and the returned values are listed on the following page.

SELECT ERROR (cont.)

Computational Errors and Returned Values:

C60	Underflow	Returned Value: 0
C61	Overflow	Returned Value: Largest positive number if actual result is positive. Largest negative number if actual result is negative.
C62	Division by zero	Returned Value: Largest positive number if number divided is positive. Largest negative number if number divided is negative.
C63	Zero raised to zero power, or zero divided by zero.	Returned Value: 0
C64	Zero raised to negative power	Returned Value: Largest positive number.
C65	Negative number X raised to non-integer power Y	Returned Value: ABS(X)^Y
C66	SQR of negative value (X)	Returned Value: SQR(ABS(X))
C67	LOG of zero	Returned Value: Largest negative number.
C68	LOG of negative value X	Returned Value: LOG(ABS(X))
C69	Argument too large for specified trig function	Returned Value: 0

The SELECT ERROR statement sets the ERROR entry in the Internal Device Table. All error messages less than or equal to this entry are suppressed. The SELECT ERROR status is returned to normal by execution of a CLEAR or LOAD RUN command.

NOTE: Error conditions which have been suppressed using the SELECT ERROR statement can not be detected using an ERROR statement until SELECT ERROR status is returned to normal. However, the occurrence of suppressed errors may be detected by inspecting ERR both before and after potential errors. A non-zero value after a calculation indicates a suppressed error occurred.

Examples:

```
0010 SELECT ERROR > 60
0010 SELECT ERROR > 69

:SELECT ERROR > 65
```

SELECT ERROR (cont.)

Compatibility Issues:

Some hardware versions support the use of optional math coprocessor hardware if appropriate interface drivers are installed. The range and precision of results near the math function singularities dealt with by SELECT ERROR may vary if this option is in use.

References:

ERROR

SELECT INPUT

General Form:

```
     SELECT INPUT {device-address  }
                  {<alpha-variable>}
```

Where:

```
     alpha-variable = an alpha-variable in which the first three bytes
                      contain an ASCII representation of a hexdigit (0-
                      9; A-F).
```

NOTE: This statement is supported for Wang compatibility reasons only and its use in new development performs no operation.

Discussion:

The SELECT INPUT statement sets the INPUT entry in the Internal Device Table to the specified device-address. This entry in the Internal Device Table performs no operation (NOP). This statement is supported for syntactical purposes only.

The slash designation used as part of the standard device-address is optional with this statement.

Examples:

```
0010 SELECT INPUT 001
:SELECT INPUT 001
```

Compatibility Issues:

In Wang 2200 Basic-2, the INPUT entry in the Internal Device Table specifies the device-address for INPUT, LINPUT, and KEYIN statements. In NPL, this entry in the Internal Device Table performs no operation. Input for these statements is always entered from the default device-address /001. This statement is supported for syntactical purposes only.

References:

SELECT LINE

General Form:

```
SELECT LINE expression
```

Discussion:

The SELECT LINE statement sets the LINE entry in the Internal Device Table to the specified value.

This entry in the device table is used to specify the maximum number of lines to be listed at one time on a screen by any LIST command or statement when the page break is in effect (LIST F option not specified).

SELECT LINE may also affect the result of PRINT AT function when this is used to erase to "end of screen". Otherwise, the output from PRINT or PRINTUSING statement in a program is not affected by the current SELECT LINE value.

The SELECT LINE value defaults to the size of the screen (24 lines) upon entering the RunTime Program.

Examples:

```
0010 SELECT LINE 67
0010 SELECT LINE X+Y-3
:SELECT LINE 16
:SELECT LINE 24
:SELECT LINE 32
```

Compatibility Issues:

On previous releases of NPL, SELECT LINE also affected LIST output directed to a printer. As of Revision 3.0 of NPL, the number of lines for LIST output directed to a printer is controlled by SELECT LISTLINE.

References:

LIST
PRINT AT
SELECT LISTLINE

SELECT LIST

General Form:

```
     SELECT LIST {device-address   }[(width)]
                 {<alpha-variable> }
```

Where:

```
     alpha-variable = an alpha-variable wherein the first three bytes
                      contain an ASCII representation of a hexdigit (0-
                      9; A-F).

     width          = a numeric-expression indicating optional line
                      width for the output device, in the range 0-255.
```

Discussion:

The SELECT LIST statement sets the LIST entry in the Internal Device Table to the specified device-address. Output generated by all of the LIST commands and its derivatives is directed to this device-address.

The width expression is used to format the LIST output to a designated line width. If the width is omitted, the last line width selected for a LIST operation is used. The default line width established at boot time is set equal to the maximum line width of the primary terminal. A specified line width of (0) indicates that line width is disregarded entirely, and that a carriage return is not issued until the entire output line has been printed, regardless of length.

Examples:

```
0010 SELECT LIST 005
0010 SELECT LIST 215(132)
:SELECT LIST <A$>
:SELECT LIST <A$>(80)
```

Compatibility Issues:

References:

SELECT LISTLINE

General Form:

```
SELECT LISTLINE expression
```

Discussion:

The SELECT LISTLINE statement sets the LISTLINE entry in the Internal Device Table to the specified value.

This entry in the device table is used to specify the maximum number of lines to be listed at one time to a printer by any LIST command or statement when the page break is in effect (LIST **S** option specified).

The SELECT LISTLINE value defaults to a value of 55 lines upon entering the RunTime Program.

The current SELECT LISTLINE value may be examined by use of the LIST DT command.

Refer to LIST, general parameters, for further information on operation of the S option.

Examples:

```
0010 SELECT LISTLINE 67
0010 SELECT LISTLINE X+Y-3
:SELECT LISTLINE 16
:SELECT LISTLINE 24
:SELECT LISTLINE 32
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater and is not supported on the Wang 2200.

References:

LIST
PRINT AT
SELECT LIST

SELECT LOG

General Form:

```
     SELECT LOG  {device-address  } [OFF]
                 {<alpha-variable>}
```

Where:

```
     alpha-variable = an alpha-variable wherein the first three bytes
                      contain an ASCII representation of a hexdigit (0-
                      9; A-F).
```

Discussion:

The SELECT LOG statement is used to select a specific output device to be used for keyboard logging operations and set the ON/OFF status of keyboard logging. The specified address is recorded in the Internal Device Table. The default SELECT LOG address is /000.

Typical use of the keyboard logging capability requires that the device-address specified is a print class device which has been defined using $DEVICE as a native operating system file.

Keyboard Logging Status:

The keyboard logging status is set to ON whenever the optional OFF keyword is not specified. Specification of the OFF keyword sets the keyboard logging status to OFF. When keyboard logging is set to ON by the SELECT LOG statement, keyboard logging to the specified device address begins immediately.

Keyboard logging status may also be modified by the operator using the Enable/Disable Keyboard Logging options on the Diag screen of the Help Processor.

NOTE: Operator options for enabling/disabling keyboard logging may be suppressed by setting on the HEX(04) bit of byte 33 of $OPTIONS. Refer to $OPTIONS for further details.

SELECT LOG (cont.)

Examining Keyboard Logging Status and Address:

The current ON/OFF status of keyboard logging is contained the HEX(02) bit of byte 19 of $MACHINE. Refer to $MACHINE for details. The current SELECT LOG address may be examined by use of the $SELECT FUNCTION:

```
X$=$SELECT(LOG)
```

In addition, LISTDT displays the current SELECT LOG address and status.

Output Generated by Keyboard Logging:

When Keyboard Logging status is on, keys pressed by the operator (with the exceptions noted below) are written to the device address specified by SELECT LOG. Special keys are expanded to the format recognized by $DEMO. Refer to Section 12.4.2 of the Programmer’s Guide for details on the "special key" format used.

Keys are not written to the specified log address in the following circumstances:

- The HELP key is not written.
- All keys pressed while in the HELP Processor are not written.
- All keys processed by a polling KEYIN are not written.

When keyboard output is logged to a native operating system ASCII file, the resulting file is suitable for use as a $DEMO script provided that polling KEYIN was not used by the application. If desired, other $DEMO features such as BOX statements or REM statements may be added to the log file before use with $DEMO by use of a text editor. Refer to $DEMO and to Chapter 12 of the Programmer’s Guide for further details on $DEMO and the contents of the Demo script file.

SELECT LOG (cont.)

Output created by the Keyboard Logging capability is treated as print class output by NPL. All features that apply to print class output do apply. This includes the ability to $OPEN the LOG device, the ability to generate NPL errors if an error occurs writing the output to a disk file using the ERR=Y $DEVICE clause, the ability to set line feed/carriage return characteristics using the ALF clause in the $DEVICE statement, and the ability to issue a (8700) $GIO microcommand to cause subsequent output to start at the start of file.

HINT: It is recommended that Keyboard Logging output be directed to a new file rather than overwriting an existing file. If an existing file is overwritten, but not completely overwritten, attempting to use this file as a $DEMO file may result in unexpected behavior when the entries left over from a previous session are encountered.

Examples:

```
0010 SELECT LOG /215
0010 SELECT LOG <A$> OFF
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

SELECT LOG is not supported on the Wang 2200.

References:

$OPTIONS - byte 33
$MACHINE - byte 19
$DEMO
$DEMO - Chapter 12 of the Programmer's Guide
Printing to ASCII text files - Chapter 5 of the NPL Supplement

SELECT ON/OFF

General Form:

```
     SELECT {ON } [{device-address} [GOSUB line-number]]
            {OFF}  {file-number   }
                   {ALERT         }
```

NOTE: This statement is supported for Wang compatibility reasons only and its use in new development performs no operation.

The compiler generates a warning when this statement is encountered.

Discussion:

The syntax of this statement is supported only for compatibility with Wang 2200 Basic-2. At execution time, this statement behaves as follows:

SELECT OFF with a device-address or ALERT but with no GOSUB clause performs no operation.

SELECT OFF with a GOSUB clause specified generates an error zero (not implemented).

All forms of SELECT ON generate an error zero (not implemented).

Examples:

Compatibility Issues:

This instruction, when executed on a Wang 2200 MVP, is used to define the interrupt status.

Under NPL, program control over interrupts is not supported.

This statement is supported only with Release 3.0 or greater.

References:

SELECT ON CLEAR

General Form:

```
SELECT ON CLEAR
```

NOTE: This statement is supported for Wang compatibility reasons only and its use in new development performs no operation.

Discussion:

The syntax of this statement is supported only for compatibility with Wang 2200 Basic-2. No operation is performed when this statement is encountered at execution time.

The compiler generates a warning when this statement is encountered.

Examples:

Compatibility Issues:

This instruction, when executed on a Wang 2200 MVP, is used to clear previously defined interrupts.

Under NPL, program control over interrupts is not supported.

This statement is supported only with Release 3.0 or greater.

References:

SELECT P

General Form:

```
SELECT P[digit]
```

Where:

```
digit = an integer in the range of 0 to 9.
```

Discussion:

The SELECT P command is used to slow output to the screen in order to render it more readable to the user (preventing screen output from rapidly scrolling by). The SELECT P command causes a pause of specified duration each time a carriage return is issued to the screen. SELECT P is also useful in debugging complex screen display operations.

The optional "digit" parameter specifies the duration of the pause to be performed before each carriage return. The digit values increment the length of the pause with SELECT P9 being the longest pause, SELECT P (or P0) being no pause. The duration of the pause is equal to the digit times one sixth of a second (i.e., SELECT P9 is a pause of 1.5 seconds, SELECT P6 is a pause of 1 second).

Any of the following resets the SELECT P value to 0:

- Execution of a SELECT P (or P0).
- Execution of a CLEAR command (with no parameters).
- RESET function.

Examples:

```
0010 SELECT P
0010 SELECT P6
:SELECT P9
:SELECT P
```

Compatibility Issues:

References:

SELECT PLOT

General Form:

```
     SELECT PLOT {device-address  }
                 {<alpha-variable>}
```

Where:

```
     alpha-variable = an alpha-variable wherein the first three bytes
                      contain an ASCII representation of a hexdigit (0-
                      9; A-F).
```

Discussion:

The SELECT PLOT statement is supported only for compatibility with Wang 2200 Basic-2. No operation is performed when this statement is encountered at execution time.

Examples:

```
0010 SELECT PLOT /415
:SELECT PLOT /415
```

Compatibility Issues:

In Wang 2200 Basic-2, the SELECT PLOT entry in the Internal Device Table specifies the device-address for PLOT operations. In NPL, this entry in the Internal Device Table performs no operation. This statement is supported for syntactical purposes only.

References:

SELECT PRINT

General Form:

```
SELECT PRINT {device-address  }[(width)]
             {<alpha-variable>}
```

Where:

```
alpha-variable =  an alpha-variable wherein the first three bytes
                  contain an ASCII representation of a hexdigit
                  (0-9; A-F).

width          =  a numeric-expression indicating optional line
                  width for the output device, in the range 0-255.
```

Discussion:

The SELECT PRINT statement sets the PRINT entry in the Internal Device Table to the specified device-address. This entry indicates the device which is to receive output from PRINT and PRINTUSING statements encountered during program execution.

Immediate mode PRINT statements are not affected by the PRINT entry selected in the Internal Device Table.

The width expression is used to format the PRINT output to a designated line width. If the width is omitted, the last line width selected for a PRINT operation is used. The default line width established at boot time is set equal to the maximum line width of the primary terminal. A specified line width of (0) indicates that line width is disregarded entirely, and that a carriage return is not issued until the entire output line has been printed, regardless of length.

Examples:

```
0010 SELECT PRINT 216(132)
0010 SELECT PRINT 205
:SELECT PRINT <A$>(64)
:SELECT PRINT 005
```

Compatibility Issues:

References:

PRINT

SELECT ROUND

General Form:

```
SELECT [NO] ROUND
```

Discussion:

The SELECT ROUND statement is used to cause rounding to take place to 14 significant digits. SELECT ROUND is the normal operation for all arithmetic operations and functions.

Specifying the NO parameter causes results to be truncated to 14 significant digits as opposed to rounding to 14 significant digits.

SELECT ROUND is the default upon entering the RunTime Program.

Examples:

```
0010 SELECT NO ROUND
0010 SELECT ROUND
:SELECT ROUND
:SELECT NO ROUND

:0010 SELECT ROUND
:0020 PRINT 2/3
:0030 SELECT NO ROUND
:0040 PRINT 2/3
:RUN
.66666666666667
.66666666666666
```

Compatibility Issues:

On a Wang 2200, SELECT NO ROUND precision is 13 significant digits.

References:

SELECT TAPE

General Form:

```
     SELECT TAPE {device-address  }
                 {<alpha-variable>}
```

Where:

```
     alpha-variable = an alpha-variable wherein the first three bytes
                      contain an ASCII representation of a hexdigit (0-
                      9; A-F).
```

NOTE: The use of this statement is not recommended. Refer to $GIO as a better alternative.

Discussion:

The SELECT TAPE statement is used to select a specific output device to be used for TAPE-class operations, and records the specified device-address in the Internal Device Table. This entry in the Internal Device Table is used as a default device for $GIO and $IF ON statements when no specific device address is given.

Examples:

```
0010 SELECT TAPE /613
:SELECT TAPE /613
```

Compatibility Issues:

References:

$GIO
$IF ON

SELECT TC

General Form:

```
     SELECT TC port-number
```

Where:

```
     port-number = {/Add            }
                   {<alpha-variable>}

     dd          = decimal value from 02 to 16
                   alpha-variable contains the the port address Add.
```

NOTE: This statement is supported for Wang compatibility reasons only and its use in new development is not supported.

Discussion:

The syntax of this statement is supported only for compatibility with Wang 2200 Basic-2. A P48 (Illegal Device Specification) error is generated when this statement is executed.

The compiler generates a warning when this statement is encountered.

Compatibility Issues:

This instruction, when executed on a Wang 2200 MVP, selects the specified port as a telecommunications port. The port must be located on a 2236MXE controller.

Under NPL, dynamic port selection for telecommunications versus terminal operation is not supported.

This statement is supported only with Release 3.0 or greater.

References:

SELECT TERMINAL

General Form:

```
     alpha-receiver = $SER
```

General Form:

```
     SELECT TERMINAL port-number
```

Where:

```
     port-number = {/Add            }
                   {<alpha-variable>}

     dd          = a decimal value from 02 to 16
                   alpha-variable contains the the port address Add.
```

NOTE: This statement is supported for Wang compatibility reasons only and its use in new development performs no operation.

Discussion:

The syntax of this statement is supported only for compatibility with Wang 2200 Basic-2. No operation is performed when this statement is encountered at execution time.

The compiler generates a warning when this statement is encountered.

Examples:

Compatibility Issues:

This instruction, when executed on a Wang 2200 MVP, selects the specified port as a terminal port.

Under NPL, dynamic port selection for telecommunications versus terminal operation is not supported.

This statement is supported only with Release 3.0 or greater.

References:

$SER

Discussion:

The $SER system variable contains the full serial number of the RunTime program currently executing as a displayable alpha-numeric value.

The complete serial number returned by $SER is also displayed on the RunTime's Gold Key diskette label.

Examples:

```
0010 X$=$SER
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

$SER is not supported on the Wang 2200.

References:

SET DATA

General Form:

```
SET DATA T [file-number,     ] file-name [,file-name]...
           [disk-address,    ]
           [<alpha-variable>,]
```

Where:

```
file-name = an alpha-variable or literal-string containing the
            name of the existing cataloged file to be scratched.
```

Discussion:

The SET DATA statement is used to set the file type of a file or files in the diskimage index to type Data. NPL currently supports two file types--programs and data files. These are represented in the diskimage index by byte 2 of the file entry for the file. This byte may have a value of (00) for data files or (80) for program files. The SET DATA command simply sets this byte to a value of (00). The corresponding SET PROGRAM statement may be used to set file type to program.

SET DATA fails with an error if any specified file on the file list is either not found or is scratched. Files listed prior to the file where the error was encountered have type set as specified.

The actual contents of the file are not affected by SET DATA except for the file trailer sector. However, operation of many NPL statements which access the file are affected if the file type is set incorrectly.

NOTE: The current file type can be determined by use of the LIMITS statement.

Examples:

```
0010 SET DATA T"START",Q$
0010 SET DATA T X$,X1$,X2$
0010 SET DATA T#Y,"SP MENU"
:SET DATA T"START"
:SET DATA T/D32,"START","SP START","SECURITY"
:SET DATA T#2,"DATA"
```

SET DATA (cont.)

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

SET DATA is not supported on the Wang 2200.

References:

SCRATCH
SET PROGRAM
UNSCRATCH
Internal structure of diskimages - Section 7.3.6 of the Programmer’s Guide

SET PROGRAM

General Form:

```
SET PROGRAM T [file-number,     ] file-name [,file-name]...
              [disk-address,    ]
              [<alpha-variable>,]
```

Where:

```
file-name = an alpha-variable or literal-string containing the
            name of the existing cataloged file to be scratched.
```

Discussion:

The SET PROGRAM statement is used to set the file type of a file or files in the diskimage index to type Program. NPL currently supports two file types--programs and data files. These are represented in the diskimage index by byte 2 of the file entry for the file. This byte may have a value of (00) for data files or (80) for program files. The SET PROGRAM command simply sets this byte to a value of (80). The corresponding SET DATA statement may be used to set file type to data.

SET PROGRAM fails with an error if any specified file on the file list is either not found or is scratched. Files listed prior to the file where the error was encountered have type set as specified.

The actual contents of the file are not affected by SET PROGRAM except for the file trailer sector. However, operation of many NPL statements which access the file are affected if the file type is set incorrectly.

NOTE: The current file type can be determined by use of the LIMITS statement.

Examples:

```
0010 SET PROGRAM T"START",Q$
0010 SET PROGRAM T X$,X1$,X2$
0010 SET PROGRAM T#Y,"SP MENU"
:SET PROGRAM T"START"
:SET PROGRAM T/D32,"START","SP START","SECURITY"
:SET PROGRAM T#2,"DATA"
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater and is not supported on the Wang 2200.

SET PROGRAM (cont.)

References:

SCRATCH
SET DATA
UNSCRATCH
Internal structure of diskimages - Section 7.3.6 of the Programmer’s Guide

SGN Function

General Form:

```
SGN (numeric-expression)
```

Discussion:

The SGN function is used to determine the sign (+,-) of a specified numeric-expression. The SGN function causes a comparison between the numeric-expression and zero. If the numeric-expression is less than zero a value of -1 is returned. If the numeric-expression is equal to zero a value of zero is returned. If the numeric-expression is greater than 0 a value of +1 is returned. This is valid wherever a numeric-expression is legal.

Examples:

```
0010 X=SGN(-12.4)
0010 PRINT SGN(.0021)
0010 Y=SGN(Y1-X)

:PRINT SGN(99.98647)
 1
:PRINT SGN(0)
0
:PRINT SGN(-152.125)
-1
```

Compatibility Issues:

References:

$SHELL

General Form:

Form 1:

```
$SHELL [literal-string] [,return-variable]
       [alpha-variable]
```

Form 2:

```
!   [literal-string]
    [alpha-variable]
```

Form 3:

```
! [command]
```

Where:

```
literal-string  = string containing native operating system com-
                  mand to be executed.

alpha-variable  = variable containing native operating system com-
                  mand to be executed.

return-variable = alpha-variable which will contain a return code
                  generated by the native operating system shell
                  upon completion of the command.
```

Discussion:

The $SHELL statement is used to effect a temporary exit from the NPL environment, and allow for interfacing with native operating system functions and programs.

Form 1 is recommended for use in programs. Form 2 can be used in programs, but not in Immediate Mode. Form 3 can be used only in Immediate Mode, and is specified without quotes surrounding the command argument.

$SHELL (cont.)

The $SHELL statement can be used to execute specific commands from the native operating system or to invoke the native operating system shell for an interactive session from within NPL. In either event, the status of the NPL program in memory at the time $SHELL is issued is unchanged when $SHELL has completed. The contents of all variables is unchanged. When the $SHELL instruction is used during program execution (as a program statement), or in Immediate Mode, program execution continues with the statement immediately following the $SHELL command (as would all other non-transfer instructions in the language).

If no literal-string or alpha-variable is specified, or the specified value is blank, then the native operating system shell is loaded for an interactive session. The user can then use any native operating system programs or functions available.

If the $SHELL statement is used to execute a specific native operating system command, control is returned automatically to the NPL program upon completion. If the $SHELL statement is used to specify an interactive session, instructions are first displayed on how to end the session when executed. When the session ends, control returns to the NPL program. In all cases, exiting from the native operating system shell returns to the exact point where NPL was originally exited.

If a "return-variable" is specified, the "return code" as specified by the shell is placed in the specified variable after performing the required command or interactive session. The value of this code depends on the native operating system and the command specified. Refer to the appropriate NPL Supplement for details.

The system performs an implicit $CLOSE on all files before $SHELL is executed.

Upon return to NPL, the current screen retains any information printed by the native operating system program. If it is desirable to retain the NPL screen, then the native operating system call function of the HELP processor may be used. Application programs should assume that any use of this statement modifies the screen display.

Examples:

```
0010 !"DIR B:"
0010 $SHELL"BACKUP",B$
0010 $SHELL A$
0010 $SHELL
:$SHELL"DIR A:"
:!DIR A:
:!cd /usr/BASIC2C; ls -l          (command under Xenix)
```

$SHELL (cont.)

Compatibility Issues:

The $SHELL statement is not valid in Wang 2200 Basic-2.

This statement is supported only with Release 2.0 or greater.

Programmed use of the $SHELL statement is highly dependent on the native operating system. The format of commands and effect of these on the operation of the RunTime depends on the operating system in use. The command to exit from the command interpreter varies--when the interactive session is started, the RunTime provides appropriate instructions on how to "exit". Availability of functions through $SHELL may require additional memory or other system resources.

References:

$SHELL - Chapter 10 of the Programmer's Guide
Platform-specific Language Features - Chapter 8 of the NPL Supplements

SIN Function

General Form:

```
SIN(numeric-expression)
```

Discussion:

The SIN function computes the value of the sine of a numeric- expression. The numeric-expression is specified as a number of radians, degrees or gradians, depending on the last executed SELECT R, D or G statement. The default at startup time is radians. This is valid wherever a numeric-expression is legal.

Examples:

```
0010 P7(R4,18) = 33*SIN(G(U2)/90)
0010 N5 = SIN(R-INT(W/90)*90)

:0010 A = 49^3+SIN((24^3)/12)
:0020 PRINT A
:RUN
 117649.821766309
```

Compatibility Issues:

Due to the use of a different algorithm, results of these functions may differ from functions evaluated on a Wang 2200. In general, however, the functions are accurate to 13 significant digits.

References:

SELECT D,R,G

$SOURCE Function

General Form:

```
receiver-variable1=$SOURCE(alpha-variable2[,alpha-variable3])

Where:

receiver-variable  = the buffer that contains the line of
                     p-code generated by $SOURCE

alpha-variable2    = ASCII text to be converted to p-code

alpha-variable3    = a long identifier table, if long
                     identifiers are to be used
```

Discussion:

The $SOURCE function is used to generate a single program line of ASCII program text from a line of NPL compatible p-code. $SOURCE used in combination with $OBJECT is useful for dynamic manipulation of NPL programs. Refer to $OBJECT for a detailed discussion of the structure of p-code files.

Under the non-interpretive RunTime program, the $SOURCE statement performs no operation.

Input to $SOURCE (alpha-variable2) must be one program line of p-code, which includes the header field. If the program line could contain long identifiers, a long identifiers table (alpha-variable3) must be specified.

The output produced is ASCII text, just as it would appear if the program line were recalled by the line editor in the Interpreter. Multi-line statements are separated by a HEX(0D) return-graphics character. One HEX(0D) always appears at the end of the line.

Should the $SOURCE function encounter difficulties decompiling (due to invalid p-code, for example) de-compilation halts at the approximate location of the problem and the output produced terminates with a "?" (instead of a HEX(0D)).

$SOURCE Function (cont.)

NOTE: Although the decompiler does many consistency checks, the fact that $SOURCE does not generate a "?" at the end of the output does not necessarily indicate that the p-code is valid.

Output from $SOURCE is limited to 1024 bytes of ASCII text, regardless of the size of the alpha-variables used.

Practical use of $SOURCE requires the dissection of a p-code file into its component p-code lines before use of $SOURCE, with suitable validity checks on the label and tests for end of file indicator. In addition, if the program uses long identifiers, the long identifiers table must be located and loaded before $SOURCE can be used. A library of functions is available to perform these functions (refer to Changes to $SOURCE functionality, Chapter 3).

Applications which use $SOURCE to locate a particular program line, and then use $OBJECT to modify that program line should follow these guidelines:

1. Always start the search at the start of the p-code file, searching program line by program line, until the desired program line is located. The length attribute stored in the header of each program line must be used to locate the next program line. Because program lines may span sectors, there is no 100% reliable way for a program to locate the start of line in any given sector other than by starting at the beginning of the program.

2. Check the length of the p-code regenerated by $OBJECT. If it is different (longer or shorter) from the original length, it is the responsibility of the program to adjust the location of all subsequent p-code lines in the p-code file. The RunTime uses the length stored in the header of the program line to determine the starting location of the next program line.

NOTE: Refer to Chapter 3, Libraries, for further information.

$SOURCE Function (cont.)

Optional Retention of Soft Carriage Returns by $SOURCE

Byte 41 of $OPTIONS may be used to specify that $SOURCE should retain soft carriage returns imbedded in the object code being translated to ASCII. The default value of HEX(00) indicates that soft carriage returns are not to be retained (this is how previous revisions worked). A value of HEX(01) indicates that soft carriage returns are to be retained.

Soft carriage returns are represented by a HEX(0D) in the ASCII string returned by $SOURCE. Retention of soft carriage returns may be desirable in order to preserve the structure of code being modified by $SOURCE/$OBJECT and is also required to preserve the syntax of lines that contain an IMAGE statement or a LINE REMARK terminated by a soft carriage return. Programs that analyze or modify the ASCII source produced by $SOURCE likely require modifications to properly handle soft carriage returns if byte 41 is set to HEX(01).

Statement separator colons that appear at the start of multi-statement lines may be changed to return spaces on $SOURCE output. This is controlled by the HEX(08) bit of $OPTIONS byte 45. When this bit is set to 0 (the default), colons are returned by $SOURCE. When this bit is set to 1, spaces are returned by $SOURCE.

Examples:

```
0010  A$=$SOURCE(B$)
0010  B$=$SOURCE(STR(X$,J,C))
0010  X$=$SOURCE(Y$)R
0010  Display$=$SOURCE(PcodeLine$,IdentifierTable$)
```

Compatibility Issues:

Versions of NPL prior to version 3.0 only permit a maximum of 512 bytes of $SOURCE output.

The form of $SOURCE in which alpha-variable3 is specified requires Revision 4.00 or later of NPL.

The $SOURCE statement is not valid in Wang 2200 Basic-2.

$SOURCE Function (cont.)

This statement requires Revision 2.00 or higher of the interpreter.

References:

$OBJECT
$OPTIONS
$KEEPREMS
$NUMBERS
Chapter 5 of the Programmers Guide
Chapter 3 of the Statements Guide

SPACE Function

General Form:

```
SPACE
```

Discussion:

The SPACE function is a numeric-function which return the amount of space available for either program text or variable definition, in bytes up to a maximum of 65488. This is valid wherever a numeric-expression is legal.

Examples:

```
0010 Y=SPACE-(X/2^6)
:PRINT SPACE
57338
```

Compatibility Issues:

The value returned by the SPACE function, in general, is substantially different for programs running under NPL compared to the same programs running on the Wang 2200 MVP. There are several reasons for this, as presented below.

- On a Wang 2200, the maximum value returned by SPACE in a 56K partition is 56220.
- Actual internal memory required by programs under NPL is at variance as compared to Wang 2200 Basic-2, due to a radically different internal program storage format (i.e., p-code).
- Actual internal memory required by variables is different. Due to a different internal format of the stack, stack overhead for variables can be larger on the NPL compiler. The following contrasts the stack overhead of the two languages.

e.g.	**variable type**	**Wang 2200**	**NPL**
		1 dimension	2 dimensions
X	numeric-scalar	4 bytes	8 bytes
X$	alpha-scalar	5 bytes	10 bytes
X()	numeric-array	6 bytes	12 bytes

X$()	alpha-array	7 bytes	14 bytes

SPACE (cont.)

In addition, memory is allocated in minimum units of 2 bytes on the 2200 and in minimum units of 16 bytes under NPL (Revision 3.0 or greater).

SPACEF and SPACEW are NPL extensions not supported on the Wang 2200.

References:

SPACEF
SPACEK
SPACEW
Dynamic Partition Size - Section 3.2 of the Programmer's Guide

SPACEF Function

General Form:

```
SPACEF
```

NOTE: SPACEF should not be used to dimension variables directly since the maximum variable size is 64K and SPACEF values may substantially exceed this. For dimensioning variables, use of the SPACE function is recommended.

Discussion:

The SPACEF function returns the amount of memory currently available in the user partition for either program text or variable definition in bytes. This is valid wherever a numeric-expression is legal.

Examples:

```
0010 A=SPACEF

:PRINT SPACEF
324782
```

Compatibility Issues:

The SPACEF function is not valid in Wang 2200 Basic-2.

This statement is supported only with Release 3.0 or greater.

References:

SPACE
SPACEW
SPACEK
Dynamic Partition Size - Section 3.2 of the Programmer's Guide

SPACEK Function

General Form:

```
SPACEK
```

NOTE: The use of this statement is not recommended. Refer to SPACEW as a better alternative.

Discussion:

The SPACEK function is a numeric function which returns the size of the user partition in number of Kilobytes (bytes/1024), up to a maximum of 61. This is valid wherever a numeric-expression is legal.

Applications which need to determine the full size of the user partition in kilobytes should use SPACEW/1024.

Due to internal dynamic memory allocation techniques, the value returned by SPACEK may vary slightly after CLEAR is executed.

Refer to Section 3.2 of the Programmer's Guide for further details on partition size.

Examples:

```
0010 W=(SPACEK-28)
:X=SPACEK

:PRINT SPACEK
 61
```

Compatibility Issues:

The SPACEK function produces results at variance with Wang Basic-2 SPACEK functions on the Wang 2200 due to the radically different partition sizes generated by NPL.

In releases prior to Revision 3.0, SPACEK never produced a value larger than 56 in NPL.

References:

SPACEF
SPACEW
SPACE
Dynamic Partition Size - Section 3.2 of the Programmer’s Guide

SPACEP Function

General Form:

```
SPACEP
```

NOTE: The use of this statement is not recommended. Refer to SPACEF as a better alternative.

Discussion:

The SPACEP function is syntactically supported for compatibility with previous releases of NPL but now returns the same value as SPACE. This is valid wherever a numeric-expression is legal.

Examples:

Compatibility Issues:

The SPACEP function is not valid in Wang 2200 Basic-2.

This statement is supported only with Release 2.0 or greater.

In NPL Revisions 2.x, SPACEP returned the amount of space available in the program segment. As of Revision 3.0, there is no longer any distinction between the program segment and the variable segment.

References:

SPACE
SPACEF
SPACEW
Dynamic Partition Size - Section 3.2 of the Programmer's Guide

SPACEV Function

General Form:

```
SPACEV
```

NOTE: The use of this statement is not recommended. Refer to SPACEF as a better alternative.

Discussion:

The SPACEV function is syntactically supported for compatibility with previous releases of NPL but now returns the same value as SPACE. This is valid wherever a numeric-expression is legal.

Examples:

Compatibility Issues:

The SPACEV function is not valid in Wang 2200 Basic-2.

This statement is supported only with Release 2.0 or greater.

In NPL Revisions 2.x, SPACEV returned the amount of space available in the variable segment. As of Revision 3.0, there is no longer any distinction between the program segment and the variable segment.

References:

SPACE
SPACEF
SPACEW
Dynamic Partition Size - Section 3.2 of the Programmer's Guide

SPACEW Function

General Form:

```
SPACEW
```

Discussion:

The SPACEW function returns the total size of the user partition in bytes. This is valid wherever a numeric-expression is legal.

Due to internal dynamic memory allocation techniques, the value returned by SPACEW may vary slightly after CLEAR is executed.

Examples:

```
0010 A=SPACEW

:PRINT SPACEW
395464
```

Compatibility Issues:

The SPACEW function is not valid in Wang 2200 Basic-2.

This statement is supported only with Release 3.0 or greater.

References:

SPACE
SPACEF
SPACEK
Dynamic Partition Size - Section 3.2 of the Programmer's Guide

SQR Function

General Form:

SQR*(numeric-expression)*

Discussion:

The SQR function computes the square root of a numeric-expression. This is valid wherever a numeric-expression is legal.

Examples:

```
0010 A=SQR(F)
0010 B,C=2+SQR(R(10,W))

:0010 A=25: B=49
:0020 PRINT SQR(A+B)
:RUN
 8.602325267042
```

Compatibility Issues:

Due to the use of different algorithms, results of these functions may differ from functions evaluated on a Wang 2200. In general, however, the functions are accurate to 13 significant digits.

References:

= statement-name (Statement Labels)

General Form:

```
=identifier
```

Where:

```
identifier =  a descriptive identifier which labels the state-
              ment.
```

Discussion:

A statement-name may serve to label a section of program with an identifier.

Statement labels may be used instead of line numbers in GOTO and GOSUB statements, as well as ON.. GOTO/GOSUB statements.

Statement labels are always private in a module; it is not possible to GOTO or GOSUB using a label in another module.

The same identifier may not be used as a statement label more than once in the same module. Statement labels which occur inside a function body are local to the function. Statement labels which occur outside all function bodies are not accessible from inside any function. The same label may be used in multiple functions within a module without conflict, or may be used in both the mainline and a function without conflict.

Statement labels are not permitted in immediate mode. An immediate mode reference to a statement label (e.g., GOSUB process_record) is executed in the context of the currently executing function (if any).

Examples:

= Statement-name (Statement Labels) (cont.)

```
0010  =Reset
0020 RecCount = 1
   : ; top of loop
   : =ProcessRecord
   :      PRINT RecCount
   :      RecCount += 1
   :      IF RecCount < 10 THEN GOTO ProcessRecord
   : =ProcessRecordEnd
0030 =InKey
   :   KEYIN key$
   :   PRINT HEXOF(key$)
   :   GOTO InKey
   : =InKeyEnd
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

GOTO
GOSUB
ON GOTO
ON GOSUB
Statement Labels - Section 4.3.2 of the Programmer's Guide

STEP

General Form:

```
STEP
```

Discussion:

The STEP statement is used to activate STEP Mode. The STEP statement can be issued in Immediate Mode or under program control. STEP Mode can also be invoked from the STEP menu item on the HELP display.

If the program is resolved, and can be continued, entering STEP from Immediate Mode displays the next statement to be executed, and sets the current LIST module to the executing module.

STEP Mode allows for "stepping" through a program one statement at a time, displaying each statement before execution. This is termed "Stepped Execution". Stepped Execution is performed by pressing the EXECUTE key while in Immediate Mode.

During Stepped Execution, if a line number range parameter has been issued (refer to STEP #), Stepped Execution is only activated within the specified line number range. By default, STEP mode is disabled for program lines located in library modules which have been loaded by INCLUDE statements. STEP mode may be explicitly enabled for these modules by:

- Selecting the module with the MODULE command.
- Enabling STEP mode for a range of lines with the STEP # command.

Stepping remains enabled on the selected range of lines in the module until the module is deleted, or a new range of step lines is selected with the above procedure. A null range of lines in the STEP # statement (STEP #1,0) disables STEP mode for all lines in the module.

Two methods are available to deactivate STEP Mode:

- Entering the CONTINUE instruction causes the program to continue normally. Refer to also CONTINUE RETURN, CONTINUE NEXT and CONTINUE LOAD for details on controlled CONTINUE statements.

STEP (cont.)

- The STEP OFF instruction deactivates STEP Mode, causing any subsequent pressing of the EXECUTE key to perform as a CONTINUE instruction.

HINT: The ability to encode a STEP statement into a program is extremely useful for debugging purposes on new or recently modified program modules.

Under the non-interpretive RunTime Program, the STEP statement performs no operation.

Examples:

```
0010 STEP
:STEP
```

Compatibility Issues:

The STEP statement is not valid in Wang 2200 Basic-2.

This statement is supported only with Release 2.0 or greater.

References:

STEP #
STEP OFF
Inspection/Modification of Program Logic - Section 6.3 of the Programmer's Guide

STEP

General Form:

```
STEP #[start[,[end]]]
```

Where:

```
start = line-number to begin STEP Mode.

end   = line-number to end STEP Mode.
```

Discussion:

The STEP # format of the STEP statement allows for designating a specified line number range within the current LIST module (termed a STEP Range) in which to limit Stepped Execution. By default, the STEP # range for the root module includes all line numbers while the default STEP # range for other modules includes no line numbers. Different STEP # values for different modules may be set by use of the MODULE command to select the desired module followed by the STEP # command to set the line # range for that module.

This allows for "stepping" through a program one statement at a time, displaying each statement before execution, but only within the specified STEP Range. Stepped Execution is performed by pressing the EXECUTE key while in Immediate Mode.

- If program control is branched into the range of line-numbers specified, STEP Mode is activated.
- If program control is branched out of the range of line-numbers specified, STEP Mode is deactivated and program execution resumes, until program control is once again passed back into the range.

If the start line number is not specified, a default of 0 is assumed. If an end line number is not specified, all lines to the end of the program are included in the range.

Selecting STEP from the HELP display changes the line number range to all lines for the executing module.

STEP # (cont.)

Two methods are available to deactivate STEP Mode:

- Entering the CONTINUE instruction causes the program to continue normally. Refer to also CONTINUE RETURN, CONTINUE NEXT and CONTINUE LOAD for details on controlled CONTINUE statements.
- The STEP OFF instruction terminates STEP Mode, causing any subsequent pressing of the EXECUTE key to perform as a CONTINUE instruction. However, if STEP (with no range parameters) is then executed, the previously specified line number range by the STEP # statement remains in effect.

The ability to encode a STEP # statement into a program is useful for debugging purposes on specific sections of new or recently modified program modules.

Under the non-interpretive RunTime Program, the STEP # statement performs no operation.

Examples:

```
:0010 STEP #
:STEP #100,1300
:STEP #100,
```

Compatibility Issues:

The STEP # statement is not valid in Wang 2200 Basic-2.

This statement is supported only with Release 2.0 or greater.

References:

STEP
STEP OFF
Inspection/Mod of Program Logic - Section 6.3 of the Programmer's Guide

STEP OFF

General Form:

```
STEP OFF
```

Discussion:

The STEP OFF statement may be used either in Immediate Mode or as a program statement.

The STEP OFF instruction deactivates STEP Mode, causing any subsequent pressing of the EXECUTE key to perform as a CONTINUE instruction. However, if STEP Mode is again activated, any previously specified line number range established by STEP # statements remain in effect in all modules.

Under the non-interpretive RunTime Program, the STEP OFF statement performs no operation.

Examples:

```
:STEP OFF
0010 STEP OFF
```

Compatibility Issues:

The STEP OFF statement is not a valid instruction in Wang 2200 Basic-2.

This statement is supported only with Release 2.0 or greater.

References:

STEP
STEP #
Inspection/Mod of Program Logic - Section 6.3 of the Programmer's Guide

STOP

General Form:

```
STOP I[text] [#]
```

Where:

```
text = any literal-string to be displayed upon execution of the
       STOP statement.

#    = causes the line number the STOP statement appears in to be
       displayed.
```

Discussion:

The STOP statement is used to invoke Immediate Mode during program execution.

Under the Interpretive RunTime Program, when a STOP statement is encountered, the program is halted, and Immediate Mode is invoked. Normal continuation of the program from that point is allowed by entering the CONTINUE command or pressing the EXECUTE key. Any other Immediate Mode commands may also be entered.

Under the Non-interpretive RunTime Program, when a STOP statement is encountered, the program is halted and the Immediate Mode prompt (":") is displayed. At this point, pressing the CANCEL key exits from the RunTime Program, pressing the EXECUTE key causes normal program continuation. Marked subroutines (DEFFN') with no parameters may also be called by pressing the appropriate SF keys. The HELP processor is also available. No other Immediate Mode functions are available.

Examples:

```
0010 STOP#
0010 STOP "This is loop line# "#
0010 STOP "Error"
```

Compatibility Issues:

Marked subroutines (DEFFN') with parameters is supported by Wang 2200 Basic-2. NPL does not support this feature.

The capabilities of Immediate Mode under NPL operate differently from those on a Wang 2200. Refer to Section 2.5.3 of the Programmer's Guide for details concerning the capabilities of Immediate Mode under NPL.

STOP (cont.)

References:

Invoking Immediate Mode - Section 2.5.2 of the Programmer's Guide
Immediate Mode Operation - Section 2.5.3 of the Programmer's Guide

STR() Function

General Form:

```
     STR(alpha-variable[,a[,b]])
                       [,,b   ]
```

Where:

```
     a = a numeric-expression specifying the first character of the
         substring in the alpha-variable.

     b = a numeric-expression specifying the number of characters in
         the substring.

         assuming d is the defined length of the alpha-variable, the
         following range conditions must be met:

              1 <= a <= d+1
              0 <= b <= d-a+1
```

Discussion:

The STRing function is an alpha function which is used to define a substring of an alpha-variable.

The STR function allows an entire alpha value or just portions of it to be examined and modified.

If the "a" parameter is omitted, the default value is 1 (the first character in the alpha-variable). If the "b" parameter is omitted, the default is equal to the rest of the alpha variable (defined length minus "a" plus 1).

A STR() function is valid wherever an alpha-variable is legal.

In some contexts, a STR function may be used to indicate that trailing spaces are important (where trailing spaces are normally not included as part of an alpha-variable).

STR() (cont.)

Examples:

```
0010 C$=STR(X$,2,4)
0010 X$=STR(A$,2)
0010 STR(Q$,2,2)=Q1$
0010 LINPUT STR(A$,,5)
0010 Y$=STR(E$(),S,N)

:0010 DIM A$20
:0020 A$="ABCDEFGHIJKLMNOP"
:0030 PRINT A$;"*"
:0040 PRINT STR(A$);"*"
:0050 PRINT STR(A$,,10);"*"
:0060 PRINT STR(A$,10);"*"
:0070 PRINT STR(A$,10,5);"*"
:RUN
ABCDEFGHIJKLMNOP*
ABCDEFGHIJKLMNOP    *
ABCDEFGHIJ*
JKLMNOP    *
JKLMN*
```

Compatibility Issues:

NPL allows a string length ("b") of 0. This is not valid in Wang 2200 Basic-2.

In Wang 2200 Basic-2, the first argument of the STR() function may not itself be a STR() function. This is legal in NPL.

String Ravel Effects

Statements which both use the value of a string and assign the value of another string encounter an interesting class of side effects when these strings overlap. Ideally, all such situations should be handled so that the result is not affected by such an overlap--the result would be the same whether the strings overlap or not. In practice, it is not generally possible to achieve this ideal, since this would require an intermediate work area big enough to store all strings whose value is required. Consequently, such operations are usually done one byte at a time, and string ravel side effects occur as a result. In general, where these effects occur, the compiler may not produce identical results to the Wang 2200 system because the compiler routines, in general, do operations two bytes at a time, if possible.

STR() (cont.)

For example, consider the following two statements:

```
A$="CAT"
STR(A$,2)=A$
```

The second statement uses both the value of A$, and sets the value of STR(A$,2). Since the strings overlap, we are dealing with the possibility of a side effect. The user might expect that the result would be the same as the statement STR(A$,2)="CAT"--which would result in a value of "CCAT" in A$.

In fact, on a Wang 2200, the result is "CCCC". To see why this happens, the user must realize that only one byte is transferred at a time from the source (A$) to the destination (STR(A$,2)), and that the number of bytes transferred is determined from the length (LEN function) of the source (in this case 3). Consequently, the assignment is equivalent to the following sequence of 1-byte transfers:

```
T$=STR(A$,1,1):STR(A$,2,1)=T$   :REM T$="C",A$="CCT"
T$=STR(A$,2,1):STR(A$,3,1)=T$   :REMT$="C",A$="CCC"
T$=STR(A$,3,1):STR(A$,4,1)=T$   :REM T$="C",A$="CCCC"
```

On the compiler, the result of the statement is quite different (the result is "CCAA"). This happens because the compiler routine transfers two bytes at a time, and transfers all bytes from the source string. So the assignment is equivalent to:

```
T$=STR(A$,1,2):STR(A$,2,2)=T$   :REM T$="CA",A$="CCA"
T$=STR(A$,3,2):STR(A$,4,2)=T$   :REM T$="A ",A$="CCAA"
T$=STR(A$,5,2):STR(A$,6,2)=T$   :REM T$="  ",A$="CCAA"
...repeat for remaining bytes   :REM no change
```

NOTE: For a large class of statements which have the potential for side effects, the problem does not in fact occur, since it makes no difference how many bytes are transferred at a time. In particular, the following, frequently used types of statements give identical results in Wang 2200 Basic-2 and in NPL:

```
STR(A$,X)=STR(A$,Y)   :REM IF X IS NOT EQUAL TO Y+1
A$=A$ & Y$            :REM FOR ANY ALPHA EXPRESSION Y$
```

For string transfer operations where the exact order of transfer is required to ensure side effects are the same as on a Wang 2200, use of the MAT COPY statement is recommended, since for this statement the exact order of transfer of bytes in the operation is guaranteed.

References:

SUB[C] Alpha-operator

General Form:

```
     alpha-receiver = [...] SUB[C] alpha-operand [...]
```

Where:

```
     alpha-operand = {literal-string  }
                     {alpha-variable  }
                     {ALL function    }
                     {BIN function    }
                     {system-variable }
```

Discussion:

The SUB alpha-operator is used to subtract the binary value of an alpha-operand from the binary value of an alpha-variable. SUB may only be used in an alpha-expression in an alpha assignment statement.

Each byte of alpha-operand is SUBtracted from each corresponding byte of the receiving alpha-variable. The SUB operation is performed from right to left, starting with the rightmost byte. If "C" immediately follows the SUB alpha-operator, then carry propagation is effected between bytes to yield full multi-byte binary number addition.

If the value of alpha-operand and the receiving alpha-variable are of different length, then the SUB algorithm implicitly extends the shorter value with leading zeroes prior to SUBtracting. If the SUB resultant is larger than the receiving alpha-variable then the extraneous high order bytes of the resultant are truncated before assignment.

NOTE: Contrary to conventional alpha-variable operations, the SUB alpha-operator operates on all bytes of an alpha-variable (either as a receiver or alpha-operand) including trailing spaces.

The SUB[C] alpha-operator is often used in conjunction with ADD[C], BIN and VAL.

SUB[C] Alpha-operator (cont.)

Examples:

```
0010 A$=SUB B$
0010 A$=SUBC ALL(01)
0010 A$=B$ SUB C$
0010 STR(A$,3,2)=SUB X$
0010 A$=SUB ’Next_Byte$(buffer$,bufpos)
0010 A$=NUM$.Hi$ SUB HEX(EF)
:0010 DIM A$2
:0020 A$=HEX(0121)
:0030 A$=SUB HEX(00FF)
:0040 PRINT "A$=";HEXOF(A$)
:RUN
A$=0122
```

Compatibility Issues:

References:

ADD
BIN
LET (Alpha Assignment)
VAL

-= numeric-expression Subtract from Variable Statement

General Form:

numeric-var -= numeric-expression

Where:

numeric-var	= a valid numeric variable (i.e., scalar or array element).
numeric-expression	= a valid numeric-expression.

Discussion:

The subtract from variable statement avoids the repetition of long variable names in common decrement uses (it is not intended to be faster than the common subtract).

This is not a numeric operator. It can only appear as a statement by itself.

NOTE: Only one variable is permitted on the left-had side of the -= , but it may be either a scalar or an array element.

Examples:

```
0010 I-=1
0010 I-=Array(X,Y)
```

Compatibility Issues:

References:

+=
LET Numeric Assignment

SWITCH Logical

General Form:

```
SWITCH
```

Discussion:

This statement declares the entry point of a logical SWITCH structure.

It is usually followed by a number of logical CASE statements. These are optionally followed by a default CASE statement. The end of the SWITCH structure is marked by a required END SWITCH statement.

When a logical SWITCH statement is executed, each of following logical CASE statements are examined, one at a time, in the order that they appear in the program. If a logical CASE statement specifies a true condition, control is transferred to the statement following that CASE statement. If no logical CASE statement specifies a true condition, and a default CASE statement is specified, control is transferred to the statement following the default CASE statement. If no logical CASE statement specifies a true condition, and no default CASE statement is specified, control is transferred to the statement following the matching END SWITCH statement.

It is possible to branch into the range of a SWITCH structure, although this is poor programming practice. If a CASE statement of any kind is encountered during execution, control is transferred to the statement following the matching END SWITCH statement.

Statements situated between a SWITCH statement and the first CASE statement are never executed. If a default CASE statement occurs in the switch structure, it must be the last CASE for the structure, or an error occurs at resolve time.

Examples:

```
0010 SWITCH
   : CASE Index<CacheIndex
   :    T=Cache(Index)
   : CASE Index<RamdiskIndex AND HasRAMDISK$="YES"
   :    T='RAMDISK_Index(Index)
   : CASE Index<VMIndex AND HasVM$="YES"
   :    T='SearchVM(Index)
   : CASE
   :    T=-1
   : END SWITCH
```

SWITCH Logical (cont.)

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

CASE (logical)
END SWITCH
Logical Constructs - Section 4.11 of NPL Programmer’s Guide

SWITCH Numeric

General Form:

```
SWITCH numeric-expression
```

Discussion:

This statement declares the entry point of a numeric SWITCH structure and defines the value of the switch expression. It is usually followed by a number of numeric CASE statements; each "case-expression" for these statements is compared to the value of the switch expression. These are optionally followed by a default CASE statement. The end of the SWITCH structure is marked by a required END SWITCH statement.

When a numeric SWITCH statement is executed, the numeric switch expression is evaluated. The value of following numeric CASE statements are examined, one at a time, in the order that they appear in the program. If a numeric CASE statement specifies an expression that is equal to the switch expression, control is transferred to the statement following that CASE statement. If no numeric CASE statement matches the switch value, and a default CASE statement is specified, control is transferred to the statement following the default CASE statement. If no numeric CASE statement matches the switch value, and no default CASE statement is specified, control is transferred to the statement following the matching END SWITCH statement.

It is possible to branch into the range of a SWITCH structure, although this is poor programming practice. If a CASE statement of any kind is encountered during execution, control is transferred to the statement following the matching END SWITCH statement.

Statements situated between a SWITCH statement and the first CASE statement are never executed. If a default CASE statement occurs in the switch structure, it must be the last CASE for the structure, or an error occurs at resolve time.

SWITCH Numeric (cont.)

Examples:

```
0010 SWITCH 12
0010 SWITCH Widget_Type
0010 SWITCH Activity_Code(Index)
:SWITCH Widget_Type
:   CASE 0
:      PRINT "Gizmos"
:   CASE 1
:      PRINT "Thingammies"
:   CASE
:      PRINT "Whatchamacallits"
:   END SWITCH
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

CASE (numeric)
END SWITCH
Logical Constructs - Section 4.11 of NPL Programmer's Guide

SWITCH String

General Form:

```
SWITCH {alpha-variable}
       {literal-string}
```

Discussion:

This statement declares the entry point of a string SWITCH structure and defines the value of the switch expression. It is usually followed by a number of string CASE statements; each "case-expression" for those statements is compared to the value of the switch expression. These are optionally followed by a default CASE statement. The end of the SWITCH structure is marked by a required END SWITCH statement.

When a string SWITCH statement is executed, the string switch value is evaluated. The value of following string CASE statements are examined, one at a time, in the order that they appear in the program. If a string CASE statement specifies a value that is equal to the switch value, control is transferred to the statement following that CASE statement. If no string CASE statement matches the switch value, and a default CASE statement is specified, control is transferred to the statement following the default CASE statement. If no string CASE statement matches the switch value, and no default CASE statement is specified, control is transferred to the statement following the matching END SWITCH statement.

It is possible to branch into the range of a SWITCH structure, although this is poor programming practice. If a CASE statement of any kind is encountered during execution, control is transferred to the statement following the matching END SWITCH statement.

Statements situated between a SWITCH statement and the first CASE statement are never executed. If a default CASE statement occurs in the switch structure, it must be the last CASE for the structure, or an error occurs at resolve time.

SWITCH String (cont.)

Examples:

```
0010 SWITCH "Alligators"
0010 SWITCH Widget_Type$
0010 SWITCH Activity_Code$(Index)
: SWITCH Widget_Type$
: CASE "Gizmos","GIZMOS"
:    PRINT 0
: CASE "Thingammies","THINGAMMIES"
:    PRINT 1
: CASE
:    PRINT "Eh?"
: END SWITCH
```

Compatibility Issues:

This statement is supported only with Release IV or greater.

References:

CASE (string)
END SWITCH
Logical Constructs - Section 4.11 of NPL Programmer’s Guide

$TAB

General Form:

Form 1

`$TAB =` *`alpha-variable`*

Form 2

`alpha-variable` `= $TAB`

Discussion:

Form 1 of the $TAB statement allows the NPL application to modify the contents of the $TAB system variable.

Form 2 allows examination of the current status of the $TAB system variable.

The $TAB system variable is 132 bytes in length. Each byte corresponds to a column position on the screen with byte 1 of $TAB corresponding to column zero, byte 2 corresponding to column 1, and so on. Values for each byte of $TAB may be either space or "T". A "T" indicates that a tab stop is defined for the corresponding column. A space indicates that no tab stop is defined for the corresponding column. Values other than space or "T" are reserved and should not be used but are treated as tab stop indicators on the current revision.

Default values for $TAB are that a "T" is present in every fourth byte starting from byte 1.

$TAB is used when TAB or SHIFT/TAB are pressed when editing a program line. Refer to TAB/SHIFT TAB, Section 5.4.12 of the Programmer's Guide, for further details.

Examples:

```
0010 X$=$TAB
0010 $TAB=X$
```

$TAB (cont.)

Compatibility Issues:

This statement is supported only with Release 3.0 or greater.

$TAB is not supported on the Wang 2200.

References:

Use of TAB/BACKTAB - Section 5.4.12 of the Programmer's Guide

TAN Function

General Form:

```
TAN (numeric-expression)
```

Discussion:

The TAN function computes the value of the tangent of a numeric-expression. This is valid wherever a numeric-expression is legal.

The numeric-expression is specified in Degrees, Radians or Gradians depending on the last executed SELECT D, R, or G statement.

Examples:

```
0010 PRINT TAN(15)
0010 N=R3+E2-TAN(90-Q1)
0010 E7=A(3,X)+TAN(15)
0010 X(M4,M5)=SIN(R4)-TAN(X(Y))
```

Compatibility Issues:

Due to the use of a different algorithm, results of these functions may differ from functions evaluated on a Wang 2200. In general, however, the functions are accurate to 13 significant digits.

References:

#TERM

General Form:

```
#TERM
```

Discussion:

The #TERM function is a numeric function which returns the terminal number of the current user. #TERM is typically used to distinguish between different users in a multi-user environment. This is valid wherever a numeric-expression is legal.

Examples:

```
0010 X=#TERM
0010 IF #TERM=2 THEN PRINT "ERROR"
```

Compatibility Issues:

The generation of #TERM is hardware-specific. Refer to the appropriate NPL Supplement for details on the hardware system.

References:

Multi-user Functions - Chapter 7 of the NPL Supplement(s)

TIME

General Form:

Form 1:

```
TIME= alpha-expression [PASSWORD {literal-string}]
                                 {alpha-variable}
```

Form 2:

```
alpha-receiver = TIME
```

Discussion:

The TIME function is a special instruction which allows inspection (form 1) and modification (form 2) of the system clock.

TIME is represented as an eight-byte alpha field in ASCII format:

Bytes 1-2	Hour (in a 24-hour format)
Bytes 3-4	Minute
Bytes 5-6	Seconds
Bytes 7-8	1/100 of a seconds, although on many machines these may always be set to 00.

Examples:

```
0010 T$=TIME
0010 T$()=TIME
0010 TIME=X$
0010 TIME="14425900"
```

Compatibility Issues:

Form 2 of the TIME statement which READS the time is fully compatible with Wang 2200 Basic-2 form.

The PASSWORD clause is required on the Wang 2200. Under NPL, the PASSWORD clause is syntactically supported for compatibility purposes, and if specified, is checked for validity. The system password is "SYSTEM" under NPL and may not be modified.

TIME (cont.)

Operation of this statement may vary on different hardware versions of NPL. Access privileges may be needed to set the system time under certain operating systems. Refer to the appropriate NPL Supplement for details.

References:

DATE
Multi-user Functions - Chapter 7 of the NPL Supplement(s)

TRACE

General Form:

```
TRACE
```

Discussion:

The TRACE statement is used to activate TRACE Mode. While TRACE Mode is on, the labels and values of all variables modified by "Trace-Sensitive" statements (refer to Section 6.3.2 of the Programmer's Guide for a list of "Trace-Sensitive" statements), and the line number of all transfer operations are displayed to the currently selected console output (CO) device as they occur. TRACE is used as a debugging tool.

NOTE: Not all variable assignments are displayed by TRACE. Only assignments by "Trace-Sensitive" statements are displayed.

The TRACE statement can be issued either in Immediate Mode or under program control. TRACE Mode is terminated by executing the TRACE OFF statement, a CLEAR statement, or the RESET function. Refer to Section 6.3 of the Programmer's Guide for details on TRACE Mode.

Under the non-interpretive RunTime program, the TRACE statement performs no operation.

TRACE output is not restricted to any specific module. Output is produced for all variable changes and transfers.

Variable assignments to function or procedure parameters which are declared as /POINTER type display the name of the referenced variable (not the name of the parameter). If the variable is declared in a module other than the current module, or in a function other than the currently executing function, the name of the module or function which declared the variable is also displayed.

TRACE (cont.)

Examples:

```
:TRACE
0010 TRACE

:0010 DIM A$1,B$16
:0020 A$="Y"
:0030 IF A$="Y" THEN 100
:0040 B$="Process Aborted"
    :            GOTO 110
:0100 B$="Process Complete"
:0110 PRINT B$

:TRACE
:RUN

A$=  "Y"                HEX(59)
TRANSFER TO 0100
B$=  "Process Complete "HEX(5072 6F63 6573 7320 436F 6D70 6C65 7465)

Process Complete
```

Compatibility Issues:

This statement is supported only with Release 2.0 or greater.

References:

TRACE OFF
TRACE #
TRACE ’
TRACE V
SELECT CO
Inspection and Modification of Program Logic - Section 6.3 of the Programmer’s Guide

TRACE OFF

General Form:

```
TRACE OFF
```

Discussion:

The TRACE OFF statement is used to deactivate TRACE Mode.

TRACE OFF can be used in conjunction with TRACE, TRACE #, TRACE ’, and TRACE V to restrict TRACE output to specified portions of a program where a suspected bug is present.

The TRACE OFF statement may be used either in Immediate Mode or under program control.

Under the non-interpretive RunTime program, the TRACE OFF statement performs no operation.

Examples:

```
:TRACE OFF
0010 TRACE OFF
0010 TRACE
   : GOSUB 1000
   : TRACE OFF
```

Compatibility Issues:

This statement is supported only with Release 2.0 or greater.

References:

TRACE
TRACE #
TRACE ’
TRACE V
Inspection and Modification of Program Logic - Section 6.3 of the Programmer’s Guide

TRACE

General Form:

```
     TRACE #[*] [[line1],[line2]]
```

Where:

```
     line1 = the lowest line-number in range specification for control-
             led TRACE Mode processing.

     line2 = the highest line-number in range specification for con-
             trolled TRACE Mode processing.
```

Discussion:

The TRACE # statement is used to turn on TRACE Mode, while suppressing all TRACE output except that associated with transfer statements to lines in the specified range. Output from TRACE Mode is displayed on the currently selected console output (CO) device. Both the line number branched to and the line number branched from are displayed by TRACE #.

The TRACE # statement causes the system to HALT program execution after executing the statement which produced the TRACE output. The "*" parameter specifies that the HALT operation is not performed.

The TRACE Mode processor can be invoked for a specified range of transfer statements by using the line1 and line2 parameters:

- If line1 is specified, all TRACE Mode output is suppressed except that associated with transfer statements to the specified line-number.
- If line1 and a comma (,) are specified, all TRACE Mode output is suppressed except that associated with transfer statements to specified line-numbers equal to or greater than the specified line-number.
- If line1 and line2 are specified, all TRACE Mode output is suppressed except that associated with transfer statements to lines within the specified range.

TRACE # (cont.)

- If only a comma (,) and line2 are specified, all TRACE output is suppressed except that associated with transfer statements to line-numbers less than or equal to the specified line-number.

TRACE # is useful when debugging programs where it is uncertain how a program branched to a particular section of code, especially when that code should not be executed.

The TRACE # statement can be issued either in Immediate Mode or under program control. TRACE Mode is terminated by implicitly specifying the TRACE OFF statement. Refer to Section 6.3 of the Programmer's Guide for details on TRACE Mode.

Under the non-interpretive RunTime program, the TRACE # statement performs no operation.

Examples:

```
:TRACE # 100
:TRACE # * 1000,2000
0010 TRACE # 950,2110
0010 TRACE # ,999
0010 TRACE # 10

:0010 IF A=0 THEN 100
:0020 PRINT A
:0030 END
:0100 A=1
   : GOTO 20

:TRACE #*
:RUN

TRANSFER FROM 0010 TO 0100
TRANSFER FROM 0100 TO 0020

1
```

Compatibility Issues:

TRACE # is not a valid instruction in Wang 2200 Basic-2.

This statement is supported only with Release 2.0 or greater

Controlled TRACE Mode (specification of range parameters) is not allowed in Wang 2200 Basic-2.

TRACE # (cont.)

References:

TRACE
TRACE OFF
SELECT CO
Inspection and Modification of Program Logic - Section 6.3 of the Programmer's Guide

TRACE '

General Form:

```
TRACE '[*] [[mark1],[mark2]]
```

Where:

```
mark1 = the lowest defined GOSUB' in range specification for con-
        trolled TRACE Mode processing.

mark2 = the highest defined GOSUB' in range specification for con-
        trolled TRACE Mode processing.
```

Discussion:

The TRACE ' statement is used to turn on TRACE Mode, while suppressing all TRACE output except that associated with transfer statements using GOSUB' to any marked subroutines in the specified range. Output from TRACE Mode is displayed on the currently selected console output (CO) device.

The TRACE ' statement causes the system to HALT program execution after executing the statement which produced the TRACE output. The "*" parameter specifies that a CONTINUE be performed automatically after each TRACE output.

The TRACE Mode processor can be invoked for a specified range of marked subroutines by using the mark1 and mark2 parameters:

- If mark1 is specified, all TRACE Mode output is suppressed except that associated with transfer statements (GOSUB') to the specified marked subroutine.
- If mark1 and a comma (,) are specified, all TRACE Mode output is suppressed except that associated with transfer statements (GOSUB') to the specified marked subroutines equal to or greater than the specified marked subroutine.
- If mark1 and mark2 are specified, all TRACE Mode output is suppressed except that associated with transfer statements (GOSUB') to marked subroutines within the specified range.

TRACE ’ (cont.)

- If only a comma (,) and mark2 are specified, all TRACE output is suppressed except that associated with transfer statements (GOSUB’) to marked subroutines less than or equal to the specified marked subroutine.

TRACE ’ is useful when debugging programs where it is uncertain how a program branched to a particular marked subroutine, especially when that subroutine should not be executed.

NOTE: LIST STACK is useful in conjunction with TRACE ’ to determine where the subroutine was called from.

The TRACE ’ statement can be issued either in Immediate Mode or under program control. TRACE Mode is terminated by implicitly specifying the TRACE OFF statement. Refer to Section 6.3 of the Programmer’s Guide for details on TRACE Mode.

Under the non-interpretive RunTime program, the TRACE ’ statement performs no operation.

TRACE’ statements are extended to include named DEFFN routines.

NOTE: In ranges of DEFFN’ names, numbered functions sort numerically but numbered functions sort lexicographically (all numbers appear before any names).

For example:

’2	**appears before ’12**	**<- numerical**
’9999	**appears before ’65535**	
’Aardvark	**appears before ’Zebra**	
’f10000	**appears before ’f9**	**<-lexical**

A LIST’ or DEFFN’ range that ends at 65535 is equivalent to ’all ranges after start value’. It is not possible to specify a range that ends exactly at 65535.

TRACE ’ (cont.)

Examples:

```
:TRACE ’ 220
:TRACE ’ * 200,240
0010 TRACE ’100
0010 TRACE ’121,200
0010 TRACE ’189,

:0010 A=1
:0020 IF A=1 THEN GOSUB ’100
:0030 GOSUB ’200
:0040 PRINT A,B,C
:0050 END
:0060 DEFFN’100
   : B=A*2
   : A=A+1
   : RETURN
:0070 DEFFN’200
   : C=B*A
   : B=B+1
   : RETURN

:TRACE ’*
:RUN

DEFFN’100
DEFFN’200

2   3              4
```

Compatibility Issues:

TRACE ’ is not a valid instruction in Wang 2200 Basic-2.

This statement is supported only with Release 2.0 or greater.

Controlled TRACE Mode (specification of range parameters) is not allowed in Wang 2200 Basic-2.

References:

TRACE
TRACE OFF
LIST STACK
SELECT CO
Inspection and Modification of Program Logic - Section 6.3 of the Programmer’s Guide

TRACE V

General Form:

```
TRACE V[*] [[var1],[var2]]
```

Where:

```
var1 = the lowest variable in range specification for controlled
       TRACE Mode processing.

var2 = the highest variable in range specification for controlled
       TRACE Mode processing.
```

Discussion:

The TRACE V statement is used to turn on TRACE Mode, while suppressing all TRACE output except that associated with variable assignments in the specified range. Output from TRACE Mode is displayed on the currently selected console output (CO) device.

The TRACE V statement causes the system to HALT program execution after executing the statement which produced the TRACE output. The "*" parameter specifies that a CONTINUE be performed automatically after each TRACE output.

TRACE Mode can be invoked for a specified range of variables ordered alphabetically by using the var1 and var2 parameters:

- If var1 parameter is specified, all TRACE Mode output is suppressed except that associated with variable assignments for the specified variable.
- If var1 and a comma (,) are specified, all TRACE Mode output is suppressed except that associated with variable assignments for variables equal to or greater than the specified variable (in ascending order).
- If var1 and var2 are specified, all TRACE Mode output is suppressed except that associated with variables within the specified range.
- If only a comma (,) and var2 are specified, all TRACE output is suppressed except that associated with variable assignments for variables less than or equal to the specified variable (in ascending order).

TRACE V (cont.)

- If no range is specified, all variables are TRACEd.
- If only one type of variable (numeric-scalar, alpha-scalar, numeric-array, alpha-array) is specified, only TRACE output concerning that type of variable is selected.

NOTE: Array variables are designated by the special format:

array-name(

For example,

A$(refers to array A$()
A(refers to array A()

TRACE V is useful when debugging programs where it is uncertain how a variable was assigned to a particular value.

The TRACE V statement can be issued either in Immediate Mode or under program control. TRACE Mode is terminated by executing the TRACE OFF statement, a CLEAR statement, or the RESET function. Refer to Section 6.3 of the Programmer's Guide for details on TRACE Mode.

Under the non-interpretive RunTime program, the TRACE V statement performs no operation.

TRACE V (cont.)

Examples:

```
:TRACE V A$
:TRACE V L(,X$
0010 TRACE V ,J1
0010 TRACE V N$(
0010 TRACE V R,
0010 TRACE V A$,J

:0010 A=1
:0020 IF A=1 THEN GOSUB '100
:0030 GOSUB '200
:0040 PRINT A,B,C
:0050 END
:0060 DEFFN'100
   : B=A*2
   : A=A+1
   : RETURN
:0070 DEFFN'200
   : C=B*A
   : B=B+1
   : RETURN

:TRACE V*
:RUN

A= 1
B= 2
A= 2
C= 4
B= 3

 2               3               4
```

Compatibility Issues:

TRACE V is not a valid instruction in Wang 2200 Basic-2.

This statement is supported only with Release 2.0 or greater.

Controlled TRACE Mode (specification of range parameters) is not allowed in Wang 2200 Basic-2.

References:

LIST V
LIST DIM
TRACE
TRACE OFF
Inspection and Modification of Program Logic - Section 6.3 of the Programmer's Guide

$TRAN

General Form:

```
    $TRAN (alpha-variable1[subs],{alpha-variable2[subs]} [hh] [R]
                                 {literal-string       }
```

Where:

```
    subs = <[s][,n]>

    hh   = two hex-digits.

    s    = a numeric expression indicating the starting position of
           the alpha-variable to use.

    n    = the number of bytes of the alpha-variable to use.  All
           bytes to the end of the alpha-variable are used if n is
           not specified.
```

Discussion:

The $TRAN statement is used to perform a character-by-character translation on all the characters in alpha-variable1. Alpha-variable2 contains a the translation table used. Typically, $TRAN is used to perform translation between character sets, i.e., ASCII and EBCDIC, lowercase to uppercase, etc.

If the "R" option is specified, alpha-variable2 must consist of pairs of characters. Each pair consists of a "TO" and a "FROM" character. If a character in arg1 is listed as one of the "FROM" characters in alpha-variable2, that character in alpha-variable1 is replaced with the corresponding "TO" character from alpha-variable2. If the character cannot be found, its value is not changed. The search in alpha-variable2 is halted when a to/from pair of HEX(2020) is encountered or the end of alpha-variable2 is reached.

If the "R" option is not specified, the binary value, plus one, of each character in alpha-variable1 is used to determine a position in alpha-variable2. The character in alpha-variable1 is then replaced with the character in the corresponding position in alpha-variable2. If alpha-variable2 is not large enough to contain the indicated position, the character in alpha-variable1 is not changed.

$TRAN (cont.)

If a mask of two hex-digits is specified, they are ANDed with each character in alpha-variable1 before the character is looked up in alpha-variable2.

Examples:

```
0010 $TRAN (Z$,HEX(0F11090A))R
0010 $TRAN (A$,Y$)0A
0010 $TRAN (B$,C$())
0010 $TRAN (STR(D$(),1,10),STR(F$(),1,16))0F R

:0010 DIM A$4,B$8
:0020 A$="ABCD"
:0030 B$="1A2B3C4D"
:0040 PRINT A$
:0050 $TRAN(A$,B$) R
:0060 PRINT A$
:RUN

ABCD
1234
```

Compatibility Issues:

References:

UNPACK

General Form:

```
     UNPACK (image) alpha-variable TO {numeric-receiver}
                                      {numeric-array   }
                                      [,{numeric-receiver}]...
                                        {numeric-array   }
```

Where:

```
     image = {[+,-] [#]...[.][#]...[^^^^]     }
             {alpha-variable containing image }
              length of image <= 254
```

Discussion:

The UNPACK statement is used to convert packed numeric information stored in an alpha-variable to one or more numeric-variables. UNPACK is normally used in conjunction with the PACK statement.

The numeric values are unpacked starting at the beginning of the specified alpha-variable. Values are converted to numeric format according to the specified image and sequentially stored in the specified numeric-variable(s) in the order listed. If numeric-arrays are specified, each element of the array is filled with one value from the alpha-variable.

Typically, the image used to unpack values is identical to the image used to pack it originally. For information on how the image determines the way packed numbers are stored, refer to the PACK statement.

An error is generated if not enough packed data is available to fill the numeric-variables or if non-BCD data appears where digits are required.

Examples:

```
0010 UNPACK(######) B$ TO A8
0010 A$="-###.##^^^^": UNPACK(A$) B$ TO A8
0010 UNPACK(-########.####) A$() TO B()
0010 UNPACK(####) STR(C$,1,6) TO A, B, C(3)
```

UNPACK (cont.)

```
:0010 DIM A(2)
:0020 PACK(+####.##) A$ FROM 12.354,-123.45,1234.56
:0030 HEXPRINT A$
:0040 UNPACK(+####.##) A$ TO A,A()
:0050 PRINT A,A(1),A(2)
:RUN

00012350101234500123456020202020

12.35          -123.45          1234.56
```

Compatibility Issues:

References:

PACK

$UNPACK

General Form:

```
     $UNPACK [({F} = unpack-specification)] alpha-variable1 TO
               {D}

                     receiver-variable [,receiver-variable]...
```

Where:

```
     unpack-specification = {alpha-variable}
                            {literal-string}

     receiver-variable    = {numeric-receiver}
                            {numeric-array   }
                            {alpha-variable  }
                            {alpha-array     }
```

Discussion:

The $UNPACK statement is used to unpack alpha-variable1 to one or more receiver-variables according to the unpack-specification. Usually, this information has been stored in alpha-variable1 using the $PACK statement. Data values are read sequentially from alpha-variable1 and is stored sequentially in the receiver-variables according to the format rules stated below. If arrays are specified in the receiver list, array-elements are filled element by element and row by row; with each element receiving one value. To treat an alpha-array as an alpha-variable, it must be specified as a STR() function.

There are three forms of the $UNPACK statement:

Delimiter Form (D parameter) each value to be unpacked is separated by a delimiter character.

$UNPACK (cont.)

Field Form (F parameter) each value occupies a specified number of bytes.

Internal Form (neither F or D is specified) data is stored in standard logical record format.

Delimiter Form

In the Delimiter form, the delimiter specification contains two bytes. The first byte is a control byte that must be set to a value from HEX(00) to HEX(03). This value is used by the $UNPACK statement to define certain unpacking rules to be followed:

00xx An error is generated if not enough fields in alpha-variable1 for variables list. Skip variables if successive delimiters occur in alpha-variable1.

01xx No error is generated if not enough fields in alpha-variable1. Skip variables if successive delimiters occur in alpha-variable1.

02xx An error is generated if not enough fields in alpha-variable1 for variables list. Ignore successive delimiters.

03xx No error is generated if not enough fields in alpha-variable1. Ignore successive delimiters.

0	0	F	F

The second byte of the delimiter specification is a delimiter character. This delimiter character separates each value in the alpha-variable to be unpacked.

When the delimiter form is used, data values may be either alpha or numeric. Alpha data values may contain any character other than the specified delimiter.

Numeric data must be stored in ASCII free format. Refer below (Field Form) for details on the internal structure of ASCII free format.

$UNPACK (cont.)

Example of Delimiter Form:

```
:0010 DIM X$32,A$5,B$5
:0020 A$="ABC": A=1.03: B$="DEFG": B=-3.2
:0030 $PACK(D=HEX(00FF)) X$ FROM A$,A,B$,B
:0040 $UNPACK(D=HEX(00FF)) X$ TO C$,C,D$,D
:0050 HEXPRINT X$ : PRINT C$,C,D$,D

:RUN

4142432020FF20312E3033FF4445464720FF2D332E32202020202020202020202020
ABC              1.03           DEFG            -3.2
```

Field Form

In the Field form, the unpack-specification contains a series of 2 byte format specifications for unpacking each value in the buffer. The first byte contains the format type. The second byte contains the length of the field.

The following types are allowed:

00xx	Skip xx bytes in alpha-variable2
10xx	ASCII free format
2dxx	ASCII integer format
3dxx	IBM display format
4dxx	IBM USASCII - 8 format
5dxx	IBM packed decimal format
6dxx	Unsigned packed decimal format
A0xx	Alpha field (length = xx bytes)
Axxx	Alpha field (length = xxx bytes)
Bd0x	Unsigned Binary format
Cd0x	Signed Binary format
Dd0x	Unsigned small endian
Ed0x	Signed small endian
Ft0x	Floating Point format

Where:

x, xx or xxx = field width in binary (x, xx or xxx > 0)

d = implied decimal position in binary

t = class of floating point format (refer below)

$UNPACK (cont.)

A	X	X	X

An individual field specification must be specified for each variable or array in the list. One field specification is specified for an array, with each element in the array being unpacked to that specification.

Alpha fields (Axxx) are treated as a character string with the length specified by xxx, allowing a field size up to 4095 bytes (4K-1).

The internal format of numeric fields is different for each of the numeric field specifications. In all cases, the length of the field is specified by byte two of the specification and the location of the implied decimal point (except for ASCII Free Format) is specified by the second hex digit of the first byte.

The $$UNPACK statement may be used to extract an entire record into a list of values. In many circumstances, it is more convenient to define a RECORD structure which defines the order and FIELD format of fields within the record. This allows a program to assign or inspect individual fields in the record buffer using numeric or string FIELD expressions, without any need to define or extract the entire list of fields. Refer to discussions in the RECORD and FIELD statements for more information.

ASCII Free Format - (10xx)

Contains a number in the format permitted for a numeric-constant. Spaces are ignored. Refer to the Glossary at the end of this guide for a description of the formats permitted for numeric-constants.

-	1.23456789	E	28

NOTE: In the case of ASCII Free Format, $UNPACK does not require an identical format to that produced with $PACK.

$UNPACK (cont.)

For example:

```
:0010 DIM X$32
:0020 $PACK(F=HEX(10101010)) X$ FROM 1.2345678901E28,-1234567890123
:0030 $UNPACK(F=HEX(10101010)) X$ TO A,B
:0040 PRINT X$: PRINT A,B

:RUN

 1.23456789E+28 -1234567890123
 1.23456789E+28 -1234567890123
```

ASCII Integer Format - (2dxx)

In this format, all digits are stored as the ASCII representation of a number. The sign is contained in byte 1 of the field. The location of the decimal point is specified implicitly by the d parameter.

-	0010065

IBM Display Format - (3dxx)

In this form, digits are stored 1 digit per byte in the format HEX(Fd) where d is the digit (0-9). The sign is stored in the high-order nibble of the last byte of the field and may be C for positive or D for negative.

F0	F0	F1	F2	D3

For example:

```
:0010 A$=HEX(F1F2D3)
:0020 $UNPACK (F=HEX(3203))A$ to A
:0030 PRINT A
:RUN

-1.23
```

IBM USASCII Format - (4dxx)

In this form, digits are stored 1 digit per byte in the format HEX(5d) where d is the digit (0-9). The sign is stored in the high order nibble of the last byte of the field and may be A for positive or B for negative.

50	50	51	52	A3

$UNPACK (cont.)

For example:

```
:0010 A$=HEX(5152A3)
:0020 $UNPACK (F=HEX(4203))A$ to A
:0030 PRINT A
:RUN
 1.23
```

IBM Packed Decimal Format - (5dxx)

In this form, digits are stored 2 digits per byte in the format HEX(dd) where d is the digit (0-9). The sign is stored in the low order nibble of the last byte of the field and may be C for positive or D for negative.

00	00	00	12	3D

For example:

```
:0010 A$=HEX(123D)
:0020 $UNPACK (F=HEX(5202))A$ to A
:0030 PRINT A
:RUN

-1.23
```

Unsigned Packed Decimal Format - (6dxx)

In this form, digits are stored 2 digits per byte in the format HEX(dd) where d is the digit (0-9). No sign is stored.

For example:

```
:0010 A$=HEX(1234)
:0020 $UNPACK (F=HEX(6202))A$ to A
:0030 PRINT A
:RUN

 12.34
```

Unsigned Binary Format (Bd0x)

Unsigned binary numbers may be specified using the format code HEX(Bd0x) where:

B	d	0	x

$UNPACK (cont.)

"B" indicates that the field is binary
"d" is a hexadecimal digit from 0 to F indicating a number of implied decimal places in the field.
"x" is the length of the field in bytes (values 1 to 5 permitted).

The following table indicates the range of values which may be stored using this format. If an implied decimal of "d" places is assumed, divide all numbers by 10^d.

Field length	**Minimum**	**Maximum**
1	0	255
2	0	65535
3	0	16777215
4	0	4294967295
5	0	1099511627775

The value returned from the field A$ is the same as that returned by VAL(A$,X)/1Ed.

Example:

```
:LIST
1000 DIM Z(4),Z1(4),X$14
   : Y=16000
   : Z(1)=196.32
   : Z(2)=100.305
   : Z(3)=1200
   : Z(4)=0
   : $PACK(F=HEX(B002B203)) X$ FROM Y,Z()
   : HEXPRINT X$
   : $UNPACK(F=HEX(B002B203)) X$ TO Y1,Z1()
   : PRINT Y1,Z1(1),Z1(2),Z1(3),Z1(4)
:RUN
3E80004CB000272E01D4C0000000R16000             196.32   100.3    1200      0
```

Signed Binary Format (Cd0x)

Signed binary numbers may be specified using the format code HEX(Cd0x) where:

C0	00	02	03

"C" indicates that the field is signed binary.
"d" is a hexadecimal digit from 0 to F indicating a number of implied decimal places in the field.
"x" is the length of the field in bytes (values 1 to 6 permitted).

$UNPACK (cont.)

The following table indicates the range of values which may be stored using this format. If an implied decimal of "d" places is assumed, divide all numbers by 10^d.

Field length	Minimum	Maximum
1	-128	127
2	-32768	32767
3	-8388608	8388607
4	-2147483648	2147483647
5	-549755813888	549755813887
6	-140737488355328	140737488355327

The value returned from the field A$ is the same as that returned by VAL(A$,-X)/1Ed.

Unsigned Small-Endian (Dd0x)

The format Dd0x is used for unsigned small-endian format, where d denotes the number of implied decimal positions and x denotes the number of bytes to be used. The number of bytes to be used may range from 1 to 5.

For example:

```
10 $PACK (F=HEX(D202)) X$ FROM 1.23
20 $UNPACK (F=HEX(D202)) X$ TO A
```

Signed Small-Endian (Ed0x)

The format Ed0x is used for signed small-endian format, where d denotes the number of implied decimal positions and x denotes the number of bytes to be used. The number of bytes to be used may range from 1 to 6.

For example:

```
10 $PACK (F=HEX(E202)) X4 FROM 1.23
20 $UNPACK (F=HEX(E202)) X$ to A
```

NOTE: "Small-endian" format is equivalent to Intel integer format.

$UNPACK (cont.)

Floating Point Format (Ft0x)

As of Revision 3.0 of NPL, field specifications of the form HEX(Ft0x) indicate the use of a floating point field "x" bytes in length. The "t" is used to distinguish between 5 classes of floating point formats.

F	2	0	4

NOTE: This differs from other previously supported numeric field format specifications, which only supported fixed point data formats and used this hex digit to indicate an implied decimal position. In addition, only a few values of the field length "x" are supported, unlike previously supported numeric field format specifications, which allow any non-zero value for the field length. The following is a summary of the supported floating point formats:

"t"	**Format**	**Valid values for "x"**
0	Wang Internal Numeric Format	8
1	NPL Internal Numeric Format	8
2	IEEE Binary Real, H-L format	4, 8
3	IEEE Binary Real, L-H format	4, 8
4	DEC VAX floating point format	4, 8
5	Sortable MAT MOVE format	2, 8

In general, where both 4 and 8 are supported as a field length, the 4-byte format corresponds to a single precision value, and the 8-byte format to a double precision value.

Purpose of the New Format

These formats have been implemented to allow NPL applications to read or generate numeric data in a format compatible with programs written in other languages such as C.

Refer to $PACK for a detailed discussion of each of the individual floating point formats.

$UNPACK (cont.)

Examples:

```
0010 Q$=HEX(A004A0105209): $UNPACK(F=Q$) Q1$() TO A$,A0$,A
0010 $UNPACK(F=HEX(5001)) T1$() TO A2
0010 $UNPACK(F=HEX(0001A00500085204)) Q1$() TO A0$,A0

:0010 DIM X$32,Y(5),A$3,B$(3)5
:0020 $PACK(F=HEX(52055205520552055205)) X$ FROM -123.45,123.456,99,-
99,34
:0030 $UNPACK(F=HEX(5205)) X$ TO Y()
:0040 HEXPRINT X$
:0050 MAT PRINT Y
:0060 $PACK(F=HEX(A00300045205A005A005A005A005)) X$ FROM
"abc",123.45,"defg","hijk","lmno"
:0070 $UNPACK(F=HEX(A00300045205A005)) X$ TO A$,C,B$()
:0080 HEXPRINT X$
:0090 PRINT A$,C,B$(1),B$(2),B$(3)

:RUN

000012345D000012345C000009900C000009900D000003400C20202020202020

-123.45
123.45
 99
-99
 34
616263345D0000000012345C646566672068696A6B206C6D6E6F202020202020
abc              123.45        defg           hijk           lmno
```

Field Format (F50x)

In this format, x represents the number of bytes in the alpha variable to be used.

The alpha representation of numeric values by this format is identical to that produced by MAT MOVE. Refer to the MAT MOVE discussion for details. The advantage of this format is that it produces alphanumeric values that can be sorted even if the numeric values contain both positive and negative values.

For example:

```
10 DIM A$8
20 $PACK(F=HEX(F508)) A$ FROM -1.234
30 $UNPACK(F=HEX(F508)) A$ to A
```

The size of the alpha variable used to store the result affects the precision of the operation. A size of eight bytes ensures full precision (13 digits). For smaller values, the precision can be calculated by the formula D= n^{*2-3} where n is the number of bytes in the alpha variable and D is the number of digits of precision.

$UNPACK (cont.)

Internal Form

The Internal form of the $UNPACK statement uses the same format used by the DATA-LOAD DC/DA statements. The values of all variables in the list are unpacked sequentially into the receiver-variable(s). If receiver-variables types do not match the data being unpacked, an error is generated. The $UNPACK operation is terminated when all data in the alpha-variable has been unpacked or all receiver-variables have been filled. Refer to Cataloged Files, Section 7.3.7 of the Programmer's Guide, for further details on the internal logical disk record format.

Examples:

```
0010 $UNPACK A$() TO B$,C(),D$(3),E
0010 $UNPACK X$ TO Z(2),Z(3),Z$,Z(4)

:0010 DIM A$(20)1
:0020 A$()=HEX(80010801012345000000008441424344FD)
:0030 $UNPACK A$() TO A,A$
:RUN

 12.345        ABCD
```

Compatibility Issues:

The unsigned binary (Bd0x) and signed binary (Cd0x) field format specifications are supported only on NPL Revision 2.1 or greater and are not supported on the Wang 2200.

The floating point (Ft0x) field format specification is supported only on NPL Revision 2.1 or greater and is not supported on the Wang 2200.

Floating point format F50x is supported only by NPL revision 3.01.11 or later.

Little-endian formats (Dxxx and Cxxx) are supported only by NPL revision 3.01.13 or later.

References:

MAT MOVE
$PACK
Internal Format of Data Files, Chapter 7 of NPL Programmer's Guide

UNSCRATCH

General Form:

```
     UNSCRATCH T [file-number,   ] file-name [,file-name]...
                 [disk-address,  ]
                 [<address-var>, ]
```

Where:

```
     file-name = an alpha-variable or literal-string containing the
                 name of the existing cataloged file to be scratched.
```

Discussion:

The UNSCRATCH statement is used to set the status of a file or files to a non-scratched condition. The purpose of this statement is to provide a convenient method to access files that were scratched in error. UNSCRATCH does not alter the contents of the file in any way except for the file trailer sector. UNSCRATCH only modifies the index entry and the file status byte of the trailer sector for the specified file.

Once UNSCRATCH has been executed, the file may be accessed normally as though it was never scratched.

No error results if the specified file is already non-scratched. However, a D82 (File not in Catalog) error results if the specified file name is not in the index of the specified diskimage.

When using UNSCRATCH, it is the responsibility of the application or programmer to make sure that no adverse effects result from the file being temporarily scratched. In particular, it should be noted that if a MOVE statement was executed while the file was in a scratched state, the file would not have been moved to the destination platter.

Examples:

```
0010 UNSCRATCH T"START",Q$
0010 UNSCRATCH T <A$>,X$,X1$,X2$
0010 UNSCRATCH T#Y,"SP MENU"
:UNSCRATCH T"START"
:UNSCRATCH T/D32,"START","SP START","SECURITY"
:UNSCRATCH T#2,"PROGRAM"
```

Compatibility Issues:

This statement is supported only with Release 3.0 or greater

UNSCRATCH (cont.)

UNSCRATCH is not supported on the Wang 2200.

References:

SAVE
LOAD
DATA SAVE DC OPEN
DATA LOAD DC OPEN
MOVE
SCRATCH

UNTIL

General Form:

```
UNTIL logical-expression
UNTIL FALSE
UNTIL TRUE
UNTIL END
```

Discussion:

The UNTIL statement marks the end of a structured REPEAT...UNTIL loop.

If the specified logical-expression is true, the loop exits and execution proceeds with the statement following the UNTIL statement. Otherwise, control is transferred to the statement following the matching REPEAT statement.

Examples:

```
0010 UNTIL FALSE
0010 UNTIL X>Y
0010 UNTIL 'ReadItem(NextItem$)<>GroupCode$
0010 UNTIL Index > MAX_INDEX OR Valid(Index)<>1

0010 REPEAT
    : X=X+X
    : PRINT X
    :UNTIL X>1000000
```

Compatibility Issues:

This statement is supported only with Release IVor greater.

References:

BREAK
REPEAT
LOOP
Logical Constructs - Section 4.11 of the NPL Programmer’s Guide

USES

General Form:

```
USES PackageIdentifier
```

Discussion:

The USES statement indicates that a PUBLIC section labeled with the specified PackageIdentifier is required by the module.

A module implicitly USES any PUBLIC sections defined within itself. If the PUBLIC section is defined by another module, an INCLUDE statement is required to specify where that module may be located.

Where named PUBLIC sections are defined, a module is only able to USE the named section if the module in which it is defined is INCLUDEd.

For example:

	(Module A)	**(Module B)**	**(Module C)**
0010	INCLUDE T "B"	INCLUDE T "C"	PUBLIC C_PRIVATE
0020	USES C_PRIVATE		END PUBLIC C_PRIVATE

The unnamed PUBLIC section of a module (if any) is implicitly USED by a module which INCLUDEs the module.

Examples:

```
0010 USES StringFunctions
0010 USES StandardColorNames
```

Compatibility Issues:

This statement is supported only with Release IVor greater.

References:

PUBLIC
INCLUDE

VAL Function

General Form:

```
VAL({alpha-variable}[,range-expression])
    {literal-string}
```

Where:

```
range-expression = a numeric-expression with result between -6
                   and +5.
```

Discussion:

The VAL function is used to convert the binary value of a specified alpha-variable or literal string to a numeric value. This is valid wherever a numeric-expression is allowed.

The range-expression of the VAL function is used to specify both the length and format of the character string to be converted. The range-expression must evaluate to a number from - 6 to + 5, otherwise an error results. If the range-expression is omitted, a value of 1 is assumed.

The absolute value of the range-expression indicates the length of the character string to be converted by VAL. A length from 0 bytes up to 6 bytes is acceptable.

If the range expression is negative, the format of the character string is assumed to be signed binary. If the range expression is positive, the format of the character string is assumed to be unsigned binary. All numbers are assumed to be stored high order-byte first.

The VAL function is the inverse of the BIN function.

VAL Function (cont.)

The following table summarizes the range of numbers which can be converted for each possible value of the range-expression.

Range Expression	**Resultant Length (bytes)**	**Type**	**Range allowed for numeric-expression**	
-6	6	signed	-140737488355328	140737488355327
-5	5	signed	-549755813886	549755813887
-4	4	signed	-2147483648	2147483647
-3	3	signed	-8388608	8388607
-2	2	signed	-32768	32767
-1	1	signed	-128	127
0	0	unsigned	0	0
1	1	unsigned	0	255
2	2	unsigned	0	65535
3	3	unsigned	0	16777215
4	4	unsigned	0	4294967295
5	5	unsigned	0	1099511627775

Examples:

```
0010 X=VAL(A$)
0010 Y=VAL(A$,4)
0010 X$=Y$(VAL(T$,2),1)

:0010 A$=HEX(923456)
:0020 PRINT VAL(A$),VAL(A$,2),VAL(A$,3)
:0030 PRINT VAL(A$,-1),VAL(A$,-2),VAL(A$,-3)

:RUN

 146          37428          9581654
-110         -28108         -7195562
```

Compatibility Issues:

In Wang 2200 Basic-2, the VAL function converts up to a maximum two byte unsigned character string only. Further, on the Wang 2200, the range-expression, if specified, must be a ",2". Numeric-expressions are not allowed as range-expressions in Wang 2200 Basic-2.

References:

BIN

VER Function

General Form:

```
     VER(data-string,pattern-string)
```

Where:

```
     data-string    = {alpha-variable}
                      {literal-string}

     pattern-string = {alpha-variable}
                      {literal-string}
```

Discussion:

The VER function is used to verify a data-string according to a specified pattern-string. This is valid wherever a numeric-expression is allowed.

The VER function performs a byte-by-byte check of the data-string against the specified pattern-string. If a character of the data-string does not belong to the set of characters defined by the corresponding byte in the specified pattern-string, the character is considered illegal and the verify operation is terminated. The VER function returns the number of data-string characters checked against the specified pattern-string before any illegal characters or the end of a string is found.

The valid character representations in the pattern-string specification are:

Pattern Character Definitions	
A	Alpha characters (Upper and lower case A-Z)
#	Numeric only (0-9)
N	Alpha or numeric (Upper and lower case A-Z
H	Hexadecimal (0-9 or A-F)
P	Packed decimal
+	+, -, or blank
X	Any character allowed
Other	Must be the same as the corresponding data-string character

VER Function (cont.)

Three occurrences cause the termination of the verification process:

- An illegal character is encountered
- End of data-string is encountered
- End of pattern-string is encountered

Examples:

```
0010 X=VER(X$,"AAAAAAAA")
0010 PRINT VER(X$,"NNNNNN+")
0010 X=VER(Y$,"NNNNNNNNNNNNNN")
0010 X=Y-VER(Y$,"HHHHHHHHH")
0010 X=VER(STR(X$,10,2),"AA")

:0010 A$="ABCD123.45"
:0020 PRINT VER(A$,"AAAAAAAAAA")
:0030 PRINT VER(A$,"AAAA###.##")
:0040 PRINT VER(A$,"NNNNNNNNNN")
:0050 PRINT VER(A$,"##########")
:0060 PRINT VER(STR(A$,5),"##########")

:RUN

 4
 10
 7
 0
 3
```

Compatibility Issues:

References:

VERIFY

General Form:

```
VERIFY T [file-number,  ][(start,end)][numeric-receiver]
         [disk-address, ]
         [<address-var>,]
```

Where:

```
start            = an expression representing the first sector to
                   be verified.

end              = an expression representing the last sector to
                   be verified.

numeric-receiver = a numeric-receiver set to the value+1 of the
                   address of the first sector that failed during
                   verify operation.  If no errors are found, re-
                   ceives a value of zero.
```

Discussion:

The VERIFY statement is used to check a diskimage or portion of a diskimage for sectors which cannot be read due to a disk hardware error or media malfunction. The VERIFY statement reads each individual sector for the range specified, checking that all information has been written correctly.

NOTE: VERIFY does not check the validity of the data; it only checks that data can be read. If the range parameters (start,end) are omitted, the entire catalog index and catalog area up to the Current End are verified.

If errors are found during the VERIFY operation, an error with the sector address is displayed on the Console Output (CO) device, and the verify operation continues. If the numeric-receiver is specified, the numeric-receiver is set to the value of the sector-address+1 containing the error and the verify operation is terminated. In this case, no error message is displayed.

VERIFY (cont.)

Examples:

```
0010 VERIFY T/D35,
0010 VERIFY T/D32,(0,1200)S
0010 VERIFY T/D10,(10,1279)
0010 VERIFY T<A$>,(10,1279)

:0010 VERIFY T#3,
:RUN

ERROR IN SECTOR 100
ERROR IN SECTOR 204
```

Compatibility Issues:

Use of the address-var parameter is supported only on NPL Revision 3.0 or greater and is not supported on the Wang 2200.

References:

SELECT CO

WEND

General Form:

```
WEND
```

Discussion:

The WEND statement marks the end of a structured WHILE...WEND loop.

NOTE: It is possible to branch into the range of a WHILE...WEND loop, although this is poor programming practice.

When the WEND statement is executed, control flows to the matching WHILE statement.

Example:

```
0010 X = 1
   :WHILE X<=10
   :   X += 1
   :   PRINT X
   : WEND
```

Compatibility Issues:

This statement is supported only with Release IVor greater.

References:

BREAK
WHILE
LOOP
WHILE/WEND - Chapter 4 of NPL Programmer’s Guide

WHILE

General Form:

```
WHILE logical-expression
WHILE FALSE
WHILE TRUE
WHILE END
```

Discussion:

The WHILE statement marks the beginning of a structured WHILE...WEND loop. It may be followed by a number of statements, which comprise the body of the loop. It must then be followed by a WEND statement.

If the specified logical-expression is false, control is transferred to the statement following the matching WEND statement. Otherwise, execution proceeds in the body of the loop.

NOTE: **It is possible to branch into the range of a WHILE...WEND loop, although this is poor programming practice.**

Examples:

```
0010 WHILE TRUE
0010 WHILE X<=Y
0010 WHILE ’ReadItem(NextItem$)=GroupCode$
0010 WHILE Index <=_MAX_INDEX AND Valid(Index)=1
```

Compatibility Issues:

This statement is supported only with Release IVor greater.

References:

BREAK
WEND
LOOP
WHILE/WEND -Chapter 4 of the NPL Programmer’s Guide

XOR Alpha-operator

General Form:

```
     alpha-receiver = [...] XOR alpha-operand [...]
```

Where:

```
     alpha-operand = { literal-string  }
                     { alpha-variable  }
                     { ALL function    }
                     { BIN function    }
                     { system-variable }
```

Discussion:

The XOR logical alpha-operator performs a logical XOR operation on the alpha-operand and the contents of the alpha-receiver, the result of which is then assigned to the alpha-receiver. The XOR alpha-operator is legal only in an alpha-expression in an alpha-assignment statement.

The XOR operation is performed on a byte-by-byte basis, moving from left to right in each field, for a number of bytes equal to the shorter of:

- The defined length of the alpha-receiver.
- The defined length of the alpha-operand (if the alpha-operand is an alpha-variable or system-variable, trailing spaces are included in the operation).

If the defined length of the alpha-operand is shorter than the defined length of the alpha-receiver, then the remaining bytes of the alpha-receiver remains unchanged (i.e., padding with spaces is **not** performed).

NOTE: In regard to the "XOR" syntactic unit, this may also appear in conditional-expressions. However, the similarity is syntactical only and its use in a conditional-expression has a completely different meaning.

XOR Alpha-operator (cont.)

Examples:

```
0010 STR(A$,4,5)=XOR B$
0010 A$=C$ XOR "0"
0010 X$=Y$ XOR Z$
0010 A$=STR(B$,4,5) XOR C$

:0010 A$=XOR buffer$.checksum$
0010: A$=XOR 'Next_Byte$(buffer$,bufpos)
:0010 DIM A$2,B$2,C$2
:0020 A$=HEX(1234)
:0030 B$=HEX(9420)
:0040 C$=A$ XOR B$
:0050 HEXPRINT C$
:RUN

8614
```

Compatibility Issues:

References

BOOL

CHAPTER 3

LIBRARY FUNCTIONS

3.1 Overview

The following chapter discusses the library functions included with the NPL Development Package.

Section 3.2 lists new files in the development package to support the library functions.

Section 3.3 discusses changes to $SOURCE functionality to support LIN’s.

Section 3.4 discusses changes to $OBJECT functionality to support LIN’s.

Section 3.5 describes field type specifications.

Section 3.6 describes field type NDM specifications.

3.2 Development Package Files for Library Functions

NPLSYS.BS2 and NPLEXAM.BS2 diskimages are now part of the development package. These diskimages include the following files:

NPLEXAM.BS2	Example use of the SOURCEIO and OBJECTIO API's.Vendors are encouraged to modify or adapt this code as required.
NPLSYS	Configuration file for examples. This defines the location of NPLSYS.BS2 library.
SRCEXAM	Example program extracts and PRINT's the ascii version of an NPL program file.
OBJEXAM	Example program generates an NPL program file from a series of string DATA items containing program lines.
SAVEPROG	Utility module defines a function 'SaveAscii that saves current RUN module source code as new\xxxxxx.SRC (replacing any existing file of that name).
NPLSYS.BS2	System library support as outlined in the reference specification. Modification of this diskimage by vendor is not recommended. Future releases of the RunTime may replace this diskimage
SOURCEIO	$SOURCE functions. This API must be used by applications that wish to obtain displayable source of Release IV NPL program files.
OBJECTIO	$OBJECT functions. This API must be used by applications that wish to generate NPL program files from displayable source at runtime.
PCKFIELD	Mnemonic names and API functions for FIELD specifications used or $PACK and NDM type fields.
BUFFERIO	Buffered I/O support. This API is used by SOURCEIO and OBJECTIO modules. The functionality of this module is not as defined in the reference specification. Direct use of the API in this module by vendors is not recommended.
PCODELBL	Common structures used by SOURCEIO and OBJECTIO when manipulating NPL program files. Direct use of the API in this module by vendors is not recommended.

3.3 Changes to $SOURCE Functionality to Support LIN’s

Some changes will be required to programs which use the previously available $SOURCE function to extract the displayable form of a program from the compiled form, when the compiled form may contain long identifier names. Attempting to use $SOURCE on programs which contain long identifiers without making these changes will result in source which terminates with an error indicator (ending in "?") when any long identifier is found.

To use the support functions for source code extraction, the library module SOURCEIO is required. Add the following line to your program:

```
INCLUDE T "SOURCEIO"
```

To support long identifier names with $SOURCE, the following additional library procedures are provided:

- ’SourceioOpenFile(FileName$8,DeviceNumber,/POINTER Stream$)

This library procedure opens the specified program name on the indicated Device. A string variable, "Stream$" is used for buffered input from the file. An internally defined RECORD is used for each open buffered file. The size of this record is returned by either the ’SourceioGetMinSize function, which is visible when SOURCEIO is INCLUDEd, or by the ’ObjectioGetMinSize function, which is visible when OBJECTIO is INCLUDEd. Applications should not modify the contents of a Stream$ variable except by calls to the API.

A non-zero NPL error code is returned if the operation cannot complete (e.g., file already exists as data file, cannot create file, etc.).

3.3.1 ’SourceioGetTableLengths

```
/POINTER Stream$,/POINTER TableLength,/POINTER MaxLineLength
```

This library routine determines the length of a long identifier name from a compiled program (opened using OpenObjectFile), and places the resulting length (in bytes) in TableLength. The size of buffer required to load the largest program line is also determined, and returned in MaxLineLength.

3.3.2 'SourceioLoadIdentifierTable

```
/POINTER Stream$,/POINTER Table$
```

This library routine initializes Table$ to contain the label and long identifier name table from a compiled program (opened using 'SourceioOpenFile). The Stream pointer is positioned to read the first program line.

3.3.3 'SourceioReadLine

```
/POINTER Stream$, /POINTER PcodeBuffer$,/POINTER End of file
```

This library routine reads the next program line into PcodeBuffer$. The size of Buffer$ should be at least MaxLineLength, as returned by 'SourceioGetTableLengths. The End of file value is set to TRUE if no more program lines are in the specified Stream.

In addition, $SOURCE will support an optional second parameter which must contain the label and long identifier table for a program, as loaded using 'SourceioLoadIdentifierTable.

3.3.4 $SOURCE

```
Pcodebuffer$[,Table$]]
```

A decompiler error (source ending in "?") will occur if $SOURCE is used to decompile a program which contains long identifiers, and no long identifier table is specified in $SOURCE, or the identifier index is invalid (exceeds size of table).

3.3.5 'SourceioCloseObjectFile

```
/POINTER Stream$
```

This library call should be done when end of file is encountered by 'SourceioReadLine, to ensure that any resources allocated to allow stream I/O are released.

3.4 $OBJECT Functionality Changes for LIN’s

In order to support long identifier names with $OBJECT, the following additional library routines are required:

- INCLUDE T "OBJECTIO"

3.4.1 ’ObjectioCreateFile

```
FileName$8,DeviceNumber,Size,/POINTER
```

This library routine initializes the specified program name on the indicated Device to be an empty program. String$ is a string field variable used for buffered output to the file. A non-zero NPL error code is returned if the operation cannot complete (e.g., file already exists as data file, cannot create file, etc.).

3.4.2 ’ObjectioClearIdentifierTable

```
/POINTER Stream$,/POINTER Table$
```

This library routine clears the specified Table$ to the initial state of the batch compiler at the start of a program, i.e., earliest label, empty long identifier table.

$OBJECT will support an optional second parameter, in which the appropriate revision label and long identifier table is incrementally maintained, e.g.:

- Output$=$OBJECT(Source$[,Table$])

3.4.3 ’ObjectioAppendLine

```
/POINTER Output$,/POINTER Stream$
```

This library routine appends the output of a $OBJECT statement to the specified Stream, which must be initialized using CreateProgramFile.

3.4.4 ’ObjectioAppendLongIdentifierTable

```
/POINTER Stream$, /POINTER Table$
```

This library routine is updates the label and long identifier name table from Table$ on the opened Stream file (which must be initialized using CreateProgramFile).

3.4.5 ’ObjectioCloseFile

```
/POINTER Stream$
```

This library call should be done after appending the identifier table to ensure that any resources allocated to allow stream I/O are released.

3.5 FIELD Type Specifications

The following section discusses the use of the FIELD type NPL specifications.

3.5.1 Defining Field Type Using $PACK Mnemonic Codes

You may use standard mnemonic field type and subtype names (instead of expressions) and little-endian binaries, if you include the standard library package "PackFormats" in the module "PCKFIELD":

```
; Module - PCKFIELD
0020 PUBLIC PackFormats
:  ;TYPES
:  DIM _PACK_ASCII_FREE_FORMAT=1
:  DIM _PACK_ASCII_INTEGER_FORMAT=2
:  DIM _PACK_IBM_DISPLAY_FORMAT=3
:  DIM _PACK_IBM_USASCII_FORMAT=4
:  DIM _PACK_IBM_PACKED_DECIMAL_FORMAT=5
:  DIM _PACK_UNSIGNED_PACKED_DECIMAL_FORMAT=6
:  DIM _PACK_ALPHA_STRING_FORMAT=10
:  DIM _PACK_UNSIGNED_BINARY_FORMAT=11
:  DIM _PACK_SIGNED_BINARY_FORMAT=12
:  DIM _PACK_UNSIGNED_BINARY_LITTLE_ENDIAN=13
:  DIM _PACK_SIGNED_BINARY_LITTLE_ENDIAN=13
:  DIM _PACK_FLOATING_POINT_FORMAT=15
:  ;SUBTYPES
:  DIM _PACK_WANG_INTERNAL_NUMERIC_FORMAT=0
:  DIM _PACK_BASIC2C_INTERNAL_NUMERIC_FORMAT=1
:  DIM _PACK_IEEE_BINARY_REAL_HL_FORMAT=2
:  DIM _PACK_IEEE_BINARY_REAL_LH_FORMAT=3
:  DIM _PACK_DEC_VAX_FLOATING_POINT_FORMAT=4
:  FUNCTION 'FieldType$(Type,Subtype,Length)/FORWARD
:  FUNCTION 'FieldAlpha$(Length)/FORWARD
:END PUBLIC PackFormats
```

Implemented as:

```
:   DIM /STATIC Type$2
:1020 FUNCTION 'FieldType$(Field_Type,Subtype,Length)/BEGINS
:     DIM Field Type$2
:     IF Field Type= PACK ALPHA STRING FORMAT
:        IF Length>4095 OR Subtype<>0 THEN RETURN ERROR (58)
:        Field Type$=HEX(A000) ADD BIN(Length,2)
:   ELSE
:        Field Type$=BIN(Field Type*16+Subtype) & BIN(Length)
:   END IF
:  RETURN (Field Type$)
: END FUNCTION 'FieldType$
: FUNCTION 'FieldAlpha$(Length)/BEGINS
:  RETURN ('FieldAlpha$(Length)/BEGINS
: END FUNCTION 'FieldAlpha$
```

Example:

```
0010 INCLUDE T/D21
0020 USES PackFormats
0030 RECRD /PUBLIC EMPLOYEE
    :    FIELD name$=HEX(A020)
    :    FIELD amount='FieldType$(_PACK_IBM_PACKED_DECIMAL_FORMAT,2,5)
    :    FIELD last_pay='FieldType$(_PACK_FLOATING_POINT_FORMAT,
    :                   _PACK_BASIC2C_INTERNAL_NUMERIC_FORMAT,8)
    : END RECORD
```

3.6 FIELD Type NDM Specifications

The following section discusses the use of the FIELD type NDM specifications.

3.6.1 Defining Field Type Using NDM Mnemonic Codes

You may use Niakwa Data Manager field type, decimal and length codes may be used instead of expressions if you include the standard library package "NDMPackFormats" is included in the module "PCKFIELD":

```
; Module - PCKFIELD
PUBLIC NDMPackFormats
   ;NDM FIELD TYPES
   DIM _NDM_ASCII_FREE_FORMAT=1
   DIM _NDM_ASCII_INTEGER_FORMAT=2
   DIM _NDM_IBM_DISPLAY_FORMAT=3
   DIM _NDM_IBM_USASCII_FORMAT=4
   DIM _NDM_IBM_PACKED_DECIMAL_FORMAT=5
   DIM _NDM_UNSIGNED_PACKED_DECIMAL_FORMAT=6
   DIM _NDM_ALPHA_FIELD=7
   DIM _NDM_UNSIGNED_BINARY_FORMAT=8
   DIM _NDM_SIGNED_BINARY_FORMAT=9
   DIM _NDM_WANG_INTERNAL_NUMERIC_FORMAT=10
   DIM _NDM_BASIC2C_INTERNAL_NUMERIC_FORMAT=11
   DIM _NDM_IEEE_BINARY_REAL_HL_FORMAT=12
   DIM _NDM_IEEE_BINARY_REAL_LH_FORMAT=13
   DIM _NDM_DEC_VAX_FLOATING_POINT_FORMAT=14
   DIM _NDM_BASIC2C_DATE=15
   DIM _NDM_PACKED_DECIMAL_DATE_YYMMDD=16
   DIM _NDM_PACKED_DECIMAL_DATE_MMDDYY=17
   DIM _NDM_PACKED_DECIMAL_DATE_YYYYMMDD=18
   DIM _NDM_ASCII_YEAR_DAYS_JULIAN_DATE_YYDDD=19
   DIM _NDM_ASCII_DAYS_JULIAN_DATE_DDDDD=20
   DIM _NDM_ALPHA_WITH_TRANSLATION=21
   FUNCTION 'NDM_FieldType$(Type,Len,Decimal)/FORWARD
END PUBLIC NDMPackFormats
```

Implemented as:

```
DIM /STATIC Type$2
FUNCTION 'NDM_FieldType$(Type,Len,Decimals)
DIM PackType,Subtype
SWITCH Type
CASE _NDM_ASCII_FREE_FORMAT
  PackType=_PACK_ASCII_FREE_FORMAT
  Subtype=0
CASE _NDM_ASCII_INTEGER_FORMAT
  PackType=_PACK_ASCII_INTEGER_FORMAT
  Subtype=Decimals
CASE _NDM_IBM_DISPLAY_FORMAT
  PackType=_PACK_IBM_DISPLAY_FORMAT
  Subtype=Decimals
CASE _NDM_IBM_USASCII_FORMAT
  PackType=_PACK_IBM_USASCII_FORMAT
  Subtype=Decimals
CASE _NDM_IBM_PACKED_DECIMAL_FORMAT
  PackType=_PACK_IBM_PACKED_DECIMAL_FORMAT
  Subtype=Decimals
CASE _NDM_UNSIGNED_PACKED_DECIMAL_FORMAT
  PackType=_PACK_UNSIGNED_PACKED_DECIMAL_FORMAT
  Subtype=Decimals
CASE _NDM_ALPHA_FIELD
  PackType=_PACK_ALPHA_STRING_FORMAT
  Subtype=0
CASE _NDM_UNSIGNED_BINARY_FORMAT
  PackType=_PACK_UNSIGNED_BINARY_FORMAT
  Subtype=Decimals
CASE _NDM_SIGNED_BINARY_FORMAT
  PackType=_PACK_SIGNED_BINARY_FORMAT
  Subtype=Decimals
CASE _NDM_WANG_INTERNAL_NUMERIC_FORMAT
  PackType=_PACK_FLOATING_POINT_FORMAT
  SubType=_PACK_WANG_INTERNAL_NUMERIC_FORMAT
  Subtype=Decimals
CASE _NDM_BASIC2C_INTERNAL_NUMERIC_FORMAT
  PackType=_PACK_FLOATING_POINT_FORMAT
  SubType=_PACK_BASIC2C_INTERNAL_NUMERIC_FORMAT
CASE _NDM_IEEE_BINARY_REAL_HL_FORMAT
  PackType=_PACK_FLOATING_POINT_FORMAT
  SubType=_PACK_IEEE_BINARY_REAL_HL_FORMAT
CASE _NDM_IEEE_BINARY_REAL_LH_FORMAT
  PackType=_PACK_FLOATING_POINT_FORMAT
  SubType=_PACK_IEEE_BINARY_REAL_LH_FORMAT
CASE _NDM_DEC_VAX_FLOATING_POINT_FORMAT
  PackType=_PACK_FLOATING_POINT_FORMAT
  SubType=_PACK_DEC_VAX_FLOATING_POINT_FORMAT
CASE _NDM_BASIC2C_DATE
  PackType=_PACK_ALPHA_STRING_FORMAT
  Subtype=0
CASE _NDM_PACKED_DECIMAL_DATE_YYMMDD
  PackType=_PACK_ALPHA_STRING_FORMAT
  Subtype=0
CASE _NDM_PACKED_DECIMAL_DATE_MMDDYY
  PackType=_PACK_ALPHA_STRING_FORMAT
  Subtype=0
CASE _NDM_PACKED_DECIMAL_DATE_YYYYMMDD
```

```
  PackType=_PACK_ALPHA_STRING_FORMAT
  Subtype=0
CASE _NDM_ASCII_YEAR_DAYS_JULIAN_DATE_YYDDD
  PackType=_PACK_ALPHA_STRING_FORMAT
  Subtype=0
CASE _NDM_ASCII_DAYS_JULIAN_DATE_DDDDD
  PackType=_PACK_ALPHA_STRING_FORMAT
  Subtype=0
CASE _NDM_ALPHA_WITH_TRANSLATION
  PackType=_PACK_ALPHA_STRING_FORMAT
  Subtype=0
CASE
  RETURN ERROR(58):; illegal field specification
END SWITCH
RETURN ('FieldType$(PackType,Subtype,Length))
```

Example:

```
0010 INCLUDE T/D21
0020 USES NDMPackFormats
0030 RECRD /PUBLIC EMPLOYEE
    :    FIELD name$=HEX(_NDM_ALPHA_FIELD,0,0)
    :    FIELD amount='FieldType$(_NDM_IBM_PACKED_DECIMAL_FORMAT,2,5)
    :    FIELD last_pay='FieldType$(_NMD_BASIC2C_INTERNAL_FORMAT,1,8)
    : END RECORD
```

APPENDIX A

RESERVED WORDS TABLE

The following terms are "reserved" and should not be used as variable names.

ABS	DIM	MIN	SPACEK
ADD	DSKIP	MOD	SPACEP
ADDC	ELSE	MODULE	SPACEV
AND	END	MOVE	SPACEW
ARC	ENDDO	NEXT	SQR
AT	ENDIF	NUM	STEP
ATN	ENDSWITCH	ON	STOP
BACKSPACE	ERR	OPEN	STR
BEG	ERROR	OR	SUBC
BIN	EXP	PACK	SWITCH
BOOL	FALSE	PLOT	TAB
BOOL0	FIELD	POS	TAN
BOOL1	FIX	PRINT	TIME
BOOL2	FN0-FN9	PRINTUSING	TO
BOOL3	FNA-FNZ	PROCEDURE	TRACE
BOOL4	FOR	PUBLIC	TRAP
BOOL5	FUNCTION	RE	TRUE
BOOL6	GOSUB	READ	TYPE
BOOL7	GOTO	RECORD	UNLINK
BOOL8	HEX	REM	UNPACK
BOOL9	HEXOF	RENAME	UNSCRATCH
BOOLA	HEXPACK	RENUMBER	UNTIL
BOOLB	HEXPRINT	REPEAT	USES
BOOLC	HEXUNPACK	RESAVE	VAL
BOOLD	IF	RESTORE	VER
BOOLE	INCLUDE	RETURN	VERIFY
BOOLF	INIT	REWIND	WEND
BOX	INPUT	RND	WHEN
BREAK	INT	ROTATE	WHILE
CASE	KEYIN	ROTATEC	XOR
CLEAR	LEN	ROUND	
CLOSE	LET	RUN	
COM	LGT	SAVE	
CONTINUE	LIMITS	SCRATCH	
CONVERT	LINK	SCREEN	
COPY	LINPUT	SEARCH	
COS	LIST	SELECT	
DATA	LOAD	SET	
DATE	LOG	SGN	
DBACKSPACE	LOOP	SIN	
DC	MAT	SKIP	
DEF	MAX	SORT	
DEFFN	MEMBER	SPACE	
DELETE	MERGE	SPACEF	

APPENDIX B

LANGUAGE COMPATIBILITY CHART

B.1 Overview

This chart is organized by specific statements and functions of the Niakwa Programming Language (NPL) language. The compatibility chart lists all Wang 2200 BASIC-2 statements (through MVP Operating System version 3.0 with some statements from higher operating system versions). In addition, where there are variances in the operation of a statement within different hardware versions of NPL, this is so indicated.

Following the list of Wang 2200 Basic-2 statements is a list of NPL extensions to the language. These statements are not supported on Wang 2200 Basic-2.

For each command, instruction and function of the language, the implementation status is indicated with the following fields:

- New--Presence of an "X" in this field indicates that the statement is not supported on the 2200. The NPL revision level the statment was implemented in is also indicated.

- Implemented (Yes, Partial, No)--These fields indicate whether the corresponding element of the 2200 BASIC-2 language is implemented in NPL. Presence of an "X" in the "Yes" column indicates that the command is implemented. Presence of an "X" in the "Partial" column indicates that the command syntax is recognized by NPL as a convenience, but that it is effectively ignored at run time. Presence of an "X" in the "No" column indicates that the command is totally unrecognized by NPL. Separate columns are provided for each of these conditions to assist in quick identification of items not implemented.

- Syntax Variance--Presence of an "X" in this field indicates the command is implemented but with some difference in syntax (no matter how slight).

- Run-Time Variance--Presence of an "X" in this field indicates the command is known in some cases to produce results at variance with 2200 BASIC-2 (no matter how slight).

- Command Extended -- Presence of an "X" in this field indicates the command has been extended with additional features beyond the specifications of 2200 BASIC-2.

  The presence of an "X" in this field for all "NEW" statements indicates the command has been extended since it's initial introduction into NPL.

- Hardware Variance--Presence of an "X" in this field indicates the results of the statement may vary, no matter how slightly, in different hardware versions of NPL.

  The presence of an "X" in this field for all "NEW" statements indicates the the results of the statement may vary in different hardware versions of NPL, since the commands initial introduction into NPL.

NOTE: Niakwa has taken due care in the preparation of this chart. To date, many thousands of Wang 2200 BASIC-2 programs of varying scope and complexity have been ported successfully to NPL. It is however the sole responsibility of the programmer to ensure that the required instruction set of the given application software is in fact satisfied by NPL through proper testing procedures.

Refer to Chapter 2 of the Statements Guide, for complete details on the function of all NPL statements including full descriptions of any incompatibilities with Wang 2200 Basic-2.

COMMAND	Implemented			Syntax Variance	RunTime Variance	Command Extended	Hardware Variance
	Y	Part	N				
ABS	X	-	-	-	-	-	-
ADD [C]	X	-	-	-	-	-	-
ALL	X	-	-	-	-	-	-
AND	X	-	-	-	-	-	-
BIN	X	-	-	-	-	X	-
BOOL	X	-	-	-	-	-	-
CLEAR	X	-	-	-	-	X	-
COM	X	-	-	-	X	X	-
COM CLEAR	X	-	-	-	-	-	-
& (Concatenation)	X	-	-	-	-	-	-
CONTINUE	X	-	-	-	-	X	-
CONVERT	X	-	-	-	-	-	-
DAC	X	-	-	-	X	-	-
DATA	X	-	-	-	-	-	-
DATE	X	-	-	X	X	-	X
DEF FN	X	-	-	-	-	-	-
DEFFN' keyboard input def.	X	-	-	-	X	-	X
DEFFN' subroutine entry point	X	-	-	X	X	X	-
DEFFN@PART	-	X	-	-	-	-	-
DIM	X	-	-	-	X	X	-
DO/ENDDO	X	-	-	X	-	X	-
DSC	X	-	-	-	X	-	-
ELSE	X	-	-	-	X	-	-
END	X	-	-	-	-	-	-
ERR	X	-	-	-	X	X	-
ERROR	X	-	-	-	X	X	-
EXEC	X	-	-	-	-	X	X
EXP	X	-	-	-	X	-	-
FIX	X	-	-	-	-	-	-
FN	X	-	-	-	-	X	-
FOR ... TO	X	-	-	-	-	-	-
GOSUB	X	-	-	-	-	X	-

COMMAND	Implemented			Syntax Variance	RunTime Variance	Command Extended	Hardware Variance
	Y	Part	N				
GOSUB’	X	-	-	X	-	X	-
GOTO	X	-	-	-	-	-	-
HALT/STEP	X	-	-	X	X	X	X
HEX	X	-	-	-	-	-	-
HEXPACK	X	-	-	-	-	-	-
HEXPRINT	X	-	-	-	-	-	-
HEXUNPACK	X	-	-	-	-	-	-
IF...THEN	-	-	-	-	-	-	-
IF END...THEN	X	-	-	-	-	-	-
Image (%)	X	-	-	-	-	X	-
INIT	X	-	-	-	-	-	-
INPUT	X	-	-	-	X	-	-
INPUT SCREEN	X	-	-	-	X	X	-
INT	X	-	-	-	-	-	-
KEYIN	X	-	-	-	X	-	-
LEN	X	-	-	-	-	-	-
LET (Alpha Assignment)	X	-	-	-	X	-	-
LET (Numeric Assignment)	X	-	-	-	X	-	-
LGT	X	-	-	-	X	-X	-
LIMITS	X	-	-	-	-	X	-
LINPUT	X	-	-	-	-	X	-
LIST (I, D, #, V, ’, T)	X	-	-	X	X	X	-
LIST DT	X	-	-	-	-	X	-
LOAD ’	-	-	-	-	-	-	-
LOG	X	-	-	-	X	-	-
MAT + (addition)	X	-	-	-	-	X	X
MAT CON (constant)	X	-	-	-	-	X	X
MAT COPY	X	-	-	-	-	-	-
MAT = (Assignment)	X	-	-	-	-	X	X
MAT IDN (identity)	X	-	-	-	-	X	X
MAT INPUT	X	-	-	-	-	-	-
MAT INV (inverse)	X	-	-	-	X	-	-

COMMAND	Implemented			Syntax Variance	RunTime Variance	Command Extended	Hardware Variance
	Y	Part	N				
MAT MERGE	X	-	-	-	X	X	X
MAT MOVE	X	-	-	-	X	X	X
MAT * (multiplication)	X	-	-	-	-	X	X
MAT PRINT	X	-	-	-	-	-	-
MAT READ	X	-	-	-	-	-	-
MAT REDIM	X	-	-	-	-	X	X
MAT ()* (scalar multiplication)	X	-	-	-	-	X	X
MAT SEARCH	X	-	-	-	-	X	X
MAT SORT	X	-	-	-	X	X	X
MAT - (subtraction)	X	-	-	-	-	X	X
MAT TRN (transposition)	X	-	-	-	-	X	X
MAT ZER (zero)	X	-	-	-	-	-	-
MAX	X	-	-	-	-	-	-
MIN	X	-	-	-	-	-	-
MOD	X	-	-	-	-	-	-
NEXT	X	-	-	-	-	-	-
NUM	X	-	-	-	-	-	-
ON ERROR	X	-	-	-	-	-	-
ON/GOSUB	X	-	-	-	-	-	-
ON/GOTO	X	-	-	-	-	-	-
ON/SELECT	X	-	-	-	-	-	-
OR	X	-	-	-	-	-	-
PACK	X	-	-	-	-	-	-
POS	X	-	-	-	-	-	-
PRINT	X	-	-	-	-	-	X
PRINT AT function	X	-	-	-	-	-	-
PRINT BOX function	X	-	-	-	X	-	X
PRINT HEXOF function	X	-	-	-	-	-	-
PRINT TAB function	X	-	-	-	-	-	-
PRINTUSING	X	-	-	-	-	X	-
PRINTUSING TO	X	-	-	-	X	-	-
READ	X	-	-	-	-	-	-

COMMAND	Implemented			Syntax Variance	RunTime Variance	Command Extended	Hardware Variance
	Y	Part	N				
REM	X	-	-	-	X	X	-
RENUMBER	X	-	-	-	-	-	-
RESET (key)	X	-	-	-	X	-	X
RESTORE [LINE]	X	-	-	-	X	-	-
RETURN	X	-	-	-	-	-	-
RETURN CLEAR	X	-	-	-	-	-	-
RND	X	-	-	-	X	-	-
ROTATE	-	-	-	-	-	-	-
ROUND	X	-	-	-	-	-	-
RUN Command	X	-	-	-	-	-	-
RUN Statement	X	-	-	-	-	X	-
SAVE BOOT	-	-	-	-	-	-	-
SCRATCH	-	-	-	-	-	-	-
SELECT DEGREES, RADIANS, GRADS	X	-	-	-	-	-	-
SELECTERROR [> error-code]	X	-	-	-	-	-	X
SELECT LINE	X	-	-	-	-	-	-
SELECT[NO] ROUND	X	-	-	-	-	-	-
SELECT PAUSE [digit]	X	-	-	-	-	-	-
SELECT CI, INPUT	X	-	-	-	X	-	-
SELECT CO, PRINT, LIST	X	-	-	-	-	-	-
SELECT PLOT, TAPE, DISK	X	-	-	-	X	-	-
SELECT# file/device-address	X	-	-	-	-	-	-
SELECT ON [dev adrs [GOSUB line]]	-	X	-	-	-	-	-
SELECT ON CLEAR	-	X	-	-	-	-	-
SELECT OFF [dev adrs [GOSUB line]]	-	X	-	-	-	-	-
SELECT DRIVER	-	X	-	-	-	-	-
SELECT TERMINAL	-	X	-	-	-	-	-
SELECT TC	-	X	-	-	-	-	-

COMMAND	Implemented			Syntax Variance	RunTime Variance	Command Extended	Hardware Variance
	Y	Part	N				
SELECT@PART	X	-	-	-	X	-	-
SGN	X	-	-	-	-	-	-
SIN	-	-	-	-	-	-	-
SPACE	X	-	-	-	X	-	-
SPACEK	X				X		
Special FN keys (function calls)	X	-	-	-	X	-	X
SQR	X	-	-	-	X	-	-
STMT NUMBER key	-	-	X	-	-	-	-
STOP	X	-	-	-	X	-	-
STR	X	-	-	-	X	X	-
SUB [C]	X	-	-	-	-	-	-
TIME	X	-	-	X	X	-	X
TRACE [OFF]	X	-	-	-	X	X	-
Trig functions: SIN, COS, TAN, ARCSIN, ARCCOS, ARCTAN, ATN	X	-	-	-	X	-	-
UNPACK	X	-	-	-	-	-	-
VAL	X	-	-	-	-	X	-
VER	X	-	-	-	-	-	-
XOR	X	-	-	-	-	-	
$ALERT	-	X	-	-	-	-	-
$BREAK	X	-	-	-	X	-	X
$CLOSE	X	-	-	-	-	X	X
$DISCONNECT	-	X	-	-	-	-	-
$FORMAT (see note below)	-	X	-	-	-	-	-
NOTE: $FORMAT syntax is accepted by the compiler (B2C) and interpreter (RTI) but is converted to a STR() function. The STR() function is functionally identical to $FORMAT.							
$GIO	X	-	-	-	X	X	-
$IF ON/OFF	X	-	-	-	X	-	-
$INIT	-	-	X	-	-	-	-
$MSG	X	-	-	-	X	-	-
$OPEN	X	-	-	-	X	X	X

COMMAND	Implemented			Syntax Variance	RunTime Variance	Command Extended	Hardware Variance
	Y	Part	N				
$PACK	X	-	-	-	-	X	-
$PSTAT	X	-	-	-	X	-	X
$RELEASE PART	-	X	-	-	-	-	-
$RELEASE TERMINAL	X	-	-	-	X	-	X
$SELECT	X	-	-	X	-	X	-
$TRAN	X	-	-	-	-	-	-
$UNPACK	X	-	-	-	-	X	-
#ID	X	-	-	-	X	-	X
#PART	X	-	-	-	-	-	X
#PI	X	-	-	-	X	-	-
#TERM	X	-	-	-	-	-	X
Disk Commands							
COPY	X	-	-	-	X	X	-
DATA LOAD BA	X	-	-	-	-	X	-
DATA LOAD BM	X	-	-	-	-	X	-
DATA LOAD DA	X	-	-	-	-	X	-
DATA LOAD DC	X	-	-	-	-	-	-
DATA LOAD DC OPEN	X	-	-	-	-	-	-
DATA SAVE BA	X	-	-	-	X	X	-
DATA SAVE BM	X	-	-	-	X	X	-
DATA SAVE DA	X	-	-	-	X	X	-
DATA SAVE DC [END]	X	-	-	-	-	X	-
DATA SAVE DC CLOSE	X	-	-	-	-	-	-
DATA SAVE DC OPEN	X	-	-	-	-	X	-
DBACKSPACE	X	-	-	-	-	-	-
DSKIP	X	-	-	-	-	-	-
IF END THEN	X	-	-	-	-	-	-
LIMITS T	X	-	-	-	-	-	-
LIMITS T (name)	X	-	-	-	-	X	-
LIST DC T	X	-	-	X	X	X	-

COMMAND	**Implemented**			**Syntax Variance**	**RunTime Variance**	**Command Extended**	**Hardware Variance**
	Y	**Part**	**N**				
LOAD Command	X	-	-	-	X	X	-
LOAD Statement	X	-	-	-	X	X	-
LOAD DA Command	X	-	-	-	X	X	-
LOAD DA Statement	X	-	-	-	X	X	-
LOAD RUN	X	-	-	-	X	X	-
MOVE	X	-	-	-	-	X	-
MOVE END	X	-	-	-	-	X	X
MOVE FILE	X	-	-	-	-	X	-
RENAME	X	-	-	-	-	X	-
RESAVE	X	-	-	-	X	X	-
SAVE	X	-	-	X	X	X	-
SAVE DA	X	-	-	X	X	X	-
SCRATCH DISK	X	-	-	-	X	X	X
VERIFY	X	-	-	-	-	X	-
$FORMAT DISK	X	-	-	-	X	X	X
Other Commands							
DATA LOAD BT	-	-	X	-	-	-	-
DATA SAVE BT	-	-	X	-	-	-	-
Other Considerations							
GLOBAL PARTITIONS	-	-	X	-	-	-	-
GLOBAL VARIABLES	X	-	-	-	-	X	-
LOCAL PRINTERS	X	-	-	-	X	-	X

Command	**New**	**Hardware Variance**
NPL Commands / System Variables (not supported on Wang 2200 Basic-2)		
+=numeric expression (Rev. 4.00)	X	-
CASE Logical (Rev. 4.00)	X	-
CASE Numeric (Rev. 4.00)	X	
CASE String (Rev. 4.00)	X	-
CASE Default (Rev. 4.00)	X	-
CONTINUE LOAD (Rev. 2.00)	X	-
CONTINUE NEXT (Rev. 2.00)	X	-
CONTINUE RETURN (Rev. 2.00)	X	-
$BOXTABLE (Rev. 1.03)	X	X
DELETE (Rev. 3.00)	X	
$DEMO (Rev. 2.00)	X	-
$DET (Rev. 3.00)	X	-
$DEVICE (Rev. 1.02)	X	X
DIM constant variable declaration (Rev. 4.00)	X	X
DIM/PUBLIC (Rev. 4.00)	X	X
DIM/RECURSIVE (Rev. 4.00)	X	X
DIM/STATIC (Rev. 4.00)	X	X
ELSE (structured) (Rev. 4.00)	X	-
$END (Rev. 1.03)	X	-
END FUNCTION (Rev. 4.00)	X	-
END IF (Rev. 4.00)	X	-
END PROCEDURE (Rev. 4.00)	X	-
END PUBLIC (Rev. 4.00)	X	-
END RECORD (Rev. 4.00)	X	-
END SWITCH (Rev. 4.00)	X	-
ERR$ (REV. 3.00)	X	-
FIELD (Rev. 4.00)	X	-
literal-string FIELD equivalent (Rev. 4.00)	X	-
numeric-expression FIELD equivalent (Rev. 4.00)	X	-
$FIELDFORMAT (Rev. 4.00)	X	-
#FIELDLENGTH (Rev. 4.00)	X	-
#FIELDSTART (Rev. 4.00)	X	-

Command	New	Hardware Variance
NPL Commands / System Variables (not supported on Wang 2200 Basic-2)		
FOR/BEGIN (Rev. 4.00)	X	-
FUNCTION (Rev. 4.00)	X	-
numeric-expression FUNCTION equivalent (Rev. IV)	X	-
literal-string FUNCTION equivalent (Rev. 4.00)	X	-
user-type-constant equivalent (Rev. 4.00)	X	-
#GOLDKEY (Rev. 1.03)	X	-
$HELP (Rev. 1.02)	X	-
$HELPINDEX (Rev. 1.03)	X	-
IF structured (Rev. 4.00)	X	-
INCLUDE (Rev. 4.00)	X	-
$KEEPREMS (Rev. 2.00)	X	-
$KEYBOARD (Rev. 1.03)	X	X
LET (numeric field assignment (Rev. 4.00)	X	-
LET (string field assignment) (Rev. 4.00)	X	-
LIMITS INDEX (Rev. 3.00)	X	-
LIST' (Rev. 3.20)	X	-
LIST DIM (Rev. 2.00)	X	-
LIST FIELD (Rev. 4.00)	X	-
LIST FUNCTION (Rev. 4.00)	X	-
LIST PROCEDURE (Rev. 4.00)	X	-
LIST PUBLIC DEFFN (Rev. 4.00)	X	-
LIST PUBLIC FIELD (Rev. 4.00)	X	-
LIST PUBLIC FUNCTION (Rev. 4.00)	X	-
LIST PUBLIC PROCEDURE (Rev. 4.00)	X	-
LIST PUBLIC RECORD (Rev. 4.00)	X	-
LIST PUBLIC V (Rev. 4.00)	X	-
LIST RECORD (Rev. 4.00)	X	-
LIST statement label (Rev. 4.00)	X	-
LIST STACK (Rev. 2.00)	X	-
LIST STACK DIM (Rev. 2.00)	X	-
LOAD BOOT (Rev. 2.00)	X	-
LOOP (Rev. 4.00)	X	-

Command	**New**	**Hardware Variance**
NPL Commands / System Variables (not supported on Wang 2200 Basic-2)		
$MACHINE (Rev. 1.03)	X	X
MODULE (Rev. 4.00)	X	-
$NAMEOF (Rev. 4.00)	X	-
$NETID (Rev. 4.00)	X	-
NEXT CLEAR (Rev. 3.00)	X	-
$NUMBERS (Rev. 2.00)	X	-
$OBJECT (Rev. 2.00)	X	-
$OPTIONS (Rev. 1.03)	X	X
$OSERR (Rev. 3.00)	X	-
PRINT SCREEN (Rev. 3.00)	X	-
PRINT TO (Rev. 3.00)	X	-
$PRINTER (Rev. 2.01)	X	-
PROCEDURE (Rev. 4.00)	X	-
call PROCEDURE by name (Rev. 4.00)	X	-
$PROGRAM (Rev. 3.00)	X	-
PUBLIC (Rev. 4.00)	X	-
READ DC (Rev. 3.00)	X	-
RECORD (Rev. 4.00)	X	-
#RECORDLENGTH (Rev. 4.00)	X	-
REM $PC (Rev. 1.02)	X	-
RENAME DEFFN' (Rev. 4.00)	X	-
RENAME FIELD (Rev. 4.00)	X	-
RENAME FUNCTION (Rev. 4.00)	X	-
RENAME PROCEDURE (Rev. 4.00)	X	-
RENAME RECORD (Rev. 4.00)	X	-
RENAME = (statement label) (Rev. 4.00)	X	-
RENAME V (Rev. 3.20)	X	-
REPEAT (Rev. 4.00)	X	-
$REV (Rev. 3.00)	X	-
SAVE BOOT (Rev. 2.00)	X	-
SELECT LISTLINE (Rev. 3.00)	X	-
SELECT LOG (Rev. 3.00)	X	-
$SER (Rev. 3.00)	X	-

Command	**New**	**Hardware Variance**
NPL Commands / System Variables (not supported on Wang 2200 Basic-2)		
SET DATA (Rev. 3.00)	X	-
SET PROGRAM (Rev. 3.00)	X	-
$SHELL (Rev. 2.00)	X	X
$SCREEN (Rev. 1.03)	X	X
$SOURCE (Rev. 2.00)	X	-
SPACEF (Rev. 3.00)	X	-
SPACEP (Rev. 2.00)	X	-
SPACEV (Rev. 2.00)	X	-
SPACEW (Rev. 3.00)	X	-
=statement-name (Statement Labels) (Rev. 4.00)	X	-
STEP (Rev. 2.00)	X	-
STEP # (Rev. 2.00)	X	-
STEP OFF (Rev. 2.00)	X	-
-=numeric-expression (Rev. 4.00)	X	-
SWITCH Logical (Rev. 4.00)	X	-
SWITCH Numeric (Rev. 4.00)	X	-
SWITCH String (Rev. 4.00)	X	-
$TAB (Rev. 3.00)	X	-
TRACE # (Rev. 2.00)	X	-
TRACE ’ (Rev. 2.00)	X	-
TRACE V (Rev. 2.00)	X	-
UNSCRATCH (Rev. 3.00)	X	-
UNTIL (Rev. 4.00)	X	-
USES (Rev. 4.00)	X	-
VERIFY (Rev. 4.00)	X	-
WEND (Rev. 4.00)	X	-
WHILE (Rev. 4.00)	X	-