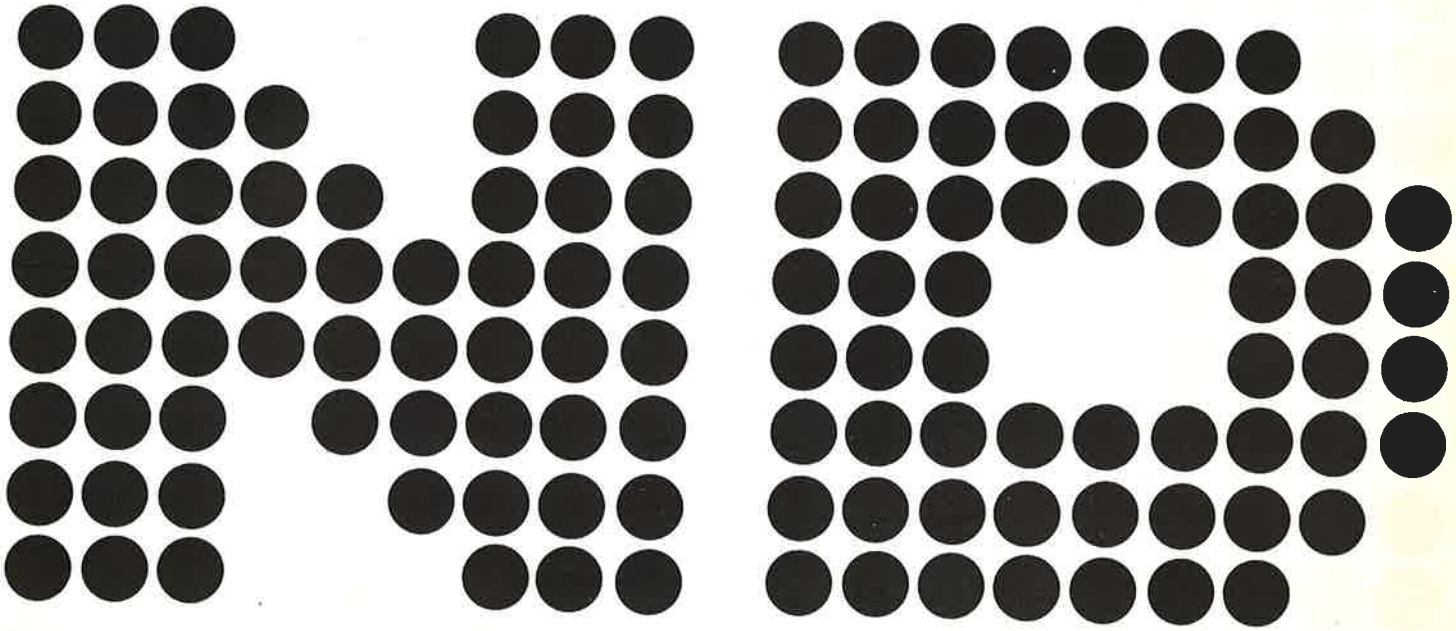
NORD DATA ENTRY SYSTEM
Reference Manual

NORSK DATA A.S



NORD DATA ENTRY SYSTEM
Reference Manual

REVISION RECORD

Revision	Notes
12/77	Original Printing
02/78	Version two

Publication No. 60.101.02

February 1978

NORSK DATA A.S.

Lørenveien 57, Postboks 163 Økern, Oslo 5, Norway

TABLE OF CONTENTS

1 Introduction

The NORD Data Entry System is designed to be a very flexible system for entering and validating data in a NORD-10 SINTRAN III computer system. The resulting data may be further processed by any of the NORD system processors such as COBOL, RPG II, SORT, FORTRAN, etc. Alternately, the data may be sent to a host machine via one of the NORD Remote Job Entry processors.

The NORD Data Entry System consists of three subsystems:

1. NORD Screen Picture Maintenance System

2. NORD Data Entry Compiler

3. NORD Data Entry Editor

The NORD Screen Picture Maintenance System is used to define screen picture formats interactively using any video display unit. Many types of field controls may be defined using this system. A description of this system will be found in the document "The NORD Screen Handling System", ND-60.088.01.

The NORD Data Entry Compiler is used when specifying the structure of a data entry application and various calculations to be performed on the entered data. The code produced by the Data Entry Compiler is used to control the way in which the Data Entry Editor is to function for a specific application. For many applications it is not necessary to use the Data Entry Compiler.

The NORD Data Entry Editor is the subsystem which is used by the operator who actually keys in the required data. The editor presents the necessary forms on the video display unit and allows the operator to enter data, validate data, inspect data and change previously entered data.

2 NORD Data Entry Compiler

2.1 Command Processor

The Data Entry Compiler is started by typing the following:

```
@DEC
NORD DATA ENTRY COMPILER V01.01

%
```

The command processor is now ready to accept commands. Whenever the command processor expects the operator to enter a command it outputs a percent sign (%). A command consists of a command name followed by zero or more parameters. Several commands, along with all required parameters, may be written on the same line.

The command name consists of one or more parts separated by hyphens ("-"). Each part of the command name may be abbreviated as long as the command can be distinguished from all other command names.

While typing commands and parameters, the editing characters ctrl-A (backspace one character), ctrl-W (backspace one word), and ctrl-Q (delete whole line) are available. Control may be returned to the command processor at any time by typing ctrl-L.

The collection of parameters is done in a standardized way as follows:

- Parameters are separated by either a comma or any number of spaces or a combination of comma and spaces.
- Parameters may be null in which case a default value is assigned.
- When a parameter is missing, as opposed to null, it is asked for and the command processor expects the user to supply the required parameter and additional parameters if desired.
- If a parameter syntax error is detected an error message is output and the parameter is asked for again.

Commands may be given directly to the SINTRAN III command processor by preceeding them with an @ sign. In this case commands to the local command processor following the SINTRAN III command are ignored.

2.2 Description of Compiler Commands

2.2.1 HELP <command name>

The HELP command lists available commands on the terminal. If <command name> is null then all available commands are listed. If <command name> is a legal command name (possibly abbreviated), the command along with its parameters is listed. If <command name> is ALL, then all commands along with their parameters are listed.

2.2.2 EXIT

The EXIT command returns control to the SINTRAN III command processor.

2.2.3 LINES <lines per page>

This command enables the user to specify the number of lines per page on the compiler listing.

2.2.4 COMPILE <source file> <list file> <application file>

This command compiles the program contained in <source file> with listing on the <list file> and output to <application file>. The default file type for <source file> and <list file> is SYMB, and for <application file> APPL. If no list file is specified, no listing is produced but error messages are printed on the terminal. If no <application file> is specified, output is written onto the system scratch file (file number 100).

2.3 Notation and Terminology

The syntax is described in a meta language in which syntactic constructs are denoted by english words or phrases, not enclosed in any special marks. These words also suggest the meaning of the construct, and are used in the accompanying description of semantics. Inside such phrases colons are substituted for spaces because space signifies concatenation of constructs. Basic symbols are enclosed within double or single quote marks, the former being the preferred form. Repetition of an item is indicated by preceeding it with a dollar sign; e.g., $(recoord:name ngth), means that the construct within parentheses may occur any (including zero) number of times. Optional constructs are enclosed within square brackets ("[" and "]"), alternatives are separated by slash ("/"), and groups of items are enclosed within parentheses. The special name ".eol" is used to denote end-of-line, and ".blank" to denote space (40 octal).

2.4 Basic Program Constituents

2.4.1 Character Set

The character set used is the ASCII character set. All non-printable characters (0-37 and 377 octal), except tab (11 octal), carriage return (15 octal), and end-of-file (27 octal) are ignored. Tabs are expanded according to the standard QED tabsetting (8, 14, 30, 40, 50, 60, 70, 80). If an end-of-file character is read, an END statement is automatically generated. Lower case characters may be used but they are considered equivalent to their upper case versions. The charecter set can be divided into letters, digits, and special characters.

```
letter = "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"/ "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";

digit = "0"/ "1"/ "2"/ "3"/ "4"/ "5"/ "6"/ "7"/ "8"/ "9";

special:character = " "/ "'"/ '"'/ "("/ ")"/ "+"/ "-"/ "*"/ "/"/
                    ","/ "."/ ":"/ ";"/ "<"/ "="/ ">";
```

The defintion of special:character contains only those characters used by the language. Other characters (e.g. "#", "%" etc.) can of course be used in alphameric strings or in comments.

2.4.2 Identifiers and Reserved Words

Identifiers are used to name global variables, fields, records, and pictures. An identifier can consist of any number of characters, the first of which must be a letter, and the remainder either a letter, digit, or the special character colon (":").

```
identifier = letter $(letter/ digit/ ":");
```

A number of identifiers have their meaning fixed in advance and are regarded as a part of the language. These are called reserved words and may not be used as programmer defined identifiers. The following is an alphabetic list of all the reserved words.

```
ACCUMULATE
ALL
AND
BY
CALCULATION
COMPUTE
COPY
DIRECT
DISPLAY
ELSE
ELSEIF
END
ENDIF
FIELD
FILE
FOR
FROM
GLOBAL
ID:FIELD
IF
IN
LOOP
MOD
NOT
OR
PICTURE:FILE
RECORD
REPEAT
RETRY
SAVE
SEQUENTIAL
STOP
STRUCTURE
TO
WHILE
WRITE
```

2.4.3 File Names

A file name denotes either a logical file or a SINTRAN III file. A logical file is a name internal to the program. A logical file must be assigned to a SINTRAN III file when the program is executed. A file name may be up to 64 characters in length. No syntax checking is performed on file names.

```
file:name = 1$64 <any character>;
```

2.4.4 Alphameric Constants

An alphameric constant consists of a sequence of characters enclosed within single or double quote marks (" or '). If the same type of quote mark which is used to enclose the string is to appear inside it, it must be written twice.

```
alpha:constant = '"' $character1 '"'/ "'" $character2 "'";

character1 = <any character except ">/ '""';

character2 = <any character except '>/ "''";
```

The following are legal alphameric constants:

```
"string", "ten o'clock", 'ten o''clock'
```

The two last examples show two different ways of writing the same string.

2.4.5 Numeric Constants

The usual decimal notation is used for numbers. A number can have a maximum of 31 digits and 9 decimal positions.

```
numeric:constant = digit $digit "." $digit;
```

The following are examples of legal numeric constants

```
101    201.03    15.
```

An integer is a decimal number which can be represented in a 16 bit word. Integers are used to represent quantities internal to the data entry system, such as lengths and array dimensions.

```
integer = digit $digit;
```

2.4.6 Data Formats

The data entry system handles data represented in five different ways. These are the following:

1. Single integer, -32768<=value<=32767. Negative numbers are represented in 2's complement format. A single integer occupies one word of storage.

2. Double integer, -2147483648<=value<=2147483647. Negative numbers are represented in 2's complement format. A double integer occupies two words of storage.

3. Byte, maximum 255 characters or 31 digits. A byte string may represent either a character string or a number. If it represents a character string of length C, it occupies INT((C+1)/2) words of storage. If it represents a number two different cases arise depending on whether it is declared as a signed field or not. If it is declared as a signed field, the rightmost position is reserved to hold the sign. Its value is space or minus. If the field is unsigned, no position is reserved for sign. A number with D digits will occupy INT((D+1)/2) words if it is unsigned and INT((D+2)/2) if it is signed.

4. "FORTRAN I5", maximum 31 digits. Each 16 bit word contains a four digit integer corresponding to four decimal digits. If one or more of the four digit integers are negative (2's complement), the number is regarded as negative. A number with D digits occupies INT((D+3)/4) words of storage.

5. BCD, Binary Coded Decimal, maximum 31 digits. Each 16 bit word represents four decimal digits, each represented in four bits. The rightmost four bits holds the sign of the number. Binary 1010 means positive and 1011 means negative. The data entry system only tests the rightmost bit to determine the sign of a BCD number. A number with D digits occupies INT(D/4)+1 words of storage.

2.5 Programs and Statements

A program consists of a sequence of statements. Statements are written one to a line and in free format. A semicolon (";") indicates that the rest of the line is to be taken as a comment. Comments are reproduced on the compiler listing but are otherwise ignored. Blank lines are treated in the same way as comment lines.

A program is divided into several sections which must appear in a certain order in the source text. The following syntax statement defines the order in which they have to appear, and which of the statements are optional or may appear several times.

```
program = picture:file:statement

          $global:statement

          $file:and:record:statement

          structure:definition

          $calculation:specification

          end:statement;
```

The following sections describe in detail the sections listed above.

2.5.1 The PICTURE:FILE Statement

The PICTURE:FILE statement specifies the SINTRAN III file which contains the definitions of the pictures to be used for this application.

picture:file:statement = "PICTURE:FILE" file:name .eol;

All of the pictures for an application must be contained within one file. The file type for the picture file is PICT, which should always be specified.

Examples:

```
PICTURE:FILE SALARY:PICT

PICTURE:FILE (SYSTEM)ABC:PICT
```

2.5.2 GLOBAL Definitions

The GLOBAL statement is used to specify all variables that are to be used in the program. Global variables are accessed in the same way as fields input from the display terminal, but they are allocated storage in a separate area. All global variables are initialized to zero when a new application is started. The global area is saved after calculations for a picture are performed. In this way it is possible to restore the state of the program at any time if execution has been interrupted or if RETRY is executed.

The syntax of the GLOBAL statement allows for repetition of groups of variables such that a variable may be referenced by its name and a set of subscripts. Such repetition can be nested to any level (dimension) and each group may contain any number of variables. All arrays are indexed from one.

```
global:statement = "GLOBAL" declaration:list .eol;

declaration:list = variable:group $("," variable:group);

variable:group = variable:definition/
                 dimension "(" variable:group $("," variable:group) ")";

variable:definition = identifier .blank length ["." decimal:positions];

length = integer;

decimal:positions = integer;
```

If the construct '"." decimal:positions' is present, the variable is declared to be numeric else alphameric. For a numeric variable the length specifies the number of significant digits. The number of decimal positions must be specified for all numeric variables, even if it is zero. For an alphameric variable the length specifies the length of the variable in bytes.

If an array is defined, the length information specifies the length of each individual element. Array elements are stored in row major order; that is, in the order (for an array declared by GLOBAL m(n(A 10.0)))

A(1,1),A(1,2),...,A(1,n),A(2,1),...,A(2,n),...,A(m,1),...,A(m,n)

Shown below are some examples of global variable and array declarations.

Examples:

```
GLOBAL ABC 10, SUM 15.2

GLOBAL 20(ABC 10, NUM 15.6)

GLOBAL 10(NAME 10, 3(NUM 4.0))
```

2.5.3 FILE and RECORD Definitions

The syntax of the FILE and RECORD statements are:

```
file:and:record:statement =
       file:statement record:statement $ record:statement;

file:statement = "FILE" file:name
                 ("DIRECT" record:length/ "SEQUENTIAL"/ "") .eol;

record:statement = "RECORD" record:list .eol;

record:list = record:definition $("," record:definition);

record:definition = identifier .blank record:length;

record:length = integer;
```

The FILE statement is used to specify each output file for the application. The name, file:name, of the file is an internal name and not the actual name of the file. The Data Entry Editor will ask for the appropriate file assignments when the OUTPUT command is executed.

An output file may be either SEQUENTIAL or DIRECT. If no type is specified, SEQUENTIAL is assumed. A SEQUENTIAL file is a normal symbolic SINTRAN III file with ASCII variable length records terminated by carriage return line feed. In the case of a SEQUENTIAL file, the specified record length is the maximum length. A DIRECT file consists of fixed size records, each record:length bytes long. DIRECT files are directly compatible with NORD RPG II. A record always starts on a word boundary. This means that if an odd number of bytes is specified an extra byte will be allocated at the end of each record. This must be taken into account when specifying the record length in another program which is to read the file.

Each FILE statement must be followed by one or more RECORD statements. This statement is used to define each possible record type for an output file. The size of the output record is specified in bytes. A buffer is allocated for each record type. The record size may not be greater than that specified in the corresponding FILE statement. If the file is a DIRECT file all records output will be of the same length, the length specified in the FILE statement. If the record length for a particular record is shorter this only means that the rest of the record is unused for this record type.

Record names are global in scope which means that they must be unique. This also means that the corresponding file can be determined from the record name alone.

Example:

```
FILE        FILE1 SEQUENTIAL
  RECORD    RECORD1 80

FILE        FILE2
  RECORD    RECORD2 80

FILE        FILE3 DIRECT 128
  RECORD    R1 10, R2 100, R3 128
  RECORD    R4 64
```

2.5.4 STRUCTURE Definitions

The STRUCTURE statement specifies that the following statements define the structure of the application. All pictures and all fields for each picture are specified. Also, if identification fields are used these are specified for each picture. The reserved word STRUCTURE may be followed by an arbitrary string which may be used for comments.

```
structure:definition = "STRUCTURE <arbitrary string> .eol
                       $picture:definition;
```

A picture definition consists of a picture name statement followed by field definition lines.

```
picture:definition = picture:name:statement $field:statement;

picture:name:statement = integer  identifier
                         ["ID:FIELD" field:definition "=" constant] .eol;

constant = alpha:constant/ numeric:constant;
```

The integer is used to refer to this picture in calculations, and the name immediately following the integer is the picture name. A picture name must not be more than eight characters in length and must be the name of one of the pictures on the picture file. If ID:FIELD is specified the following field:definition define the identification field of this picture. This field must be the first field of the picture (uppermost and leftmost). The rules which apply to the definition of this field are otherwise the same as for other fields, as described below. The constant following the field:definition specifies the value which will cause this picture to be selected.

Either all or none of the pictures may have an identification field. If identification fields are specified, these fields must all have the same attributes.

After the picture:name:statement follows zero or more field:statements which have the syntax:

```
field:statement = "FIELD" field:declaration:list .eol;

field:declaration:list = field:group $("," field:group);

field:group = field:definition/
              dimension "(" field:group $("," field:group) ")";

field:definition = identifier .blank [length ["." decimal:positions]];
```

This is essentially the same syntax as for global variables, except that the length information may be omitted. If it is omitted the required field attributes are read from the picture file. If the length information is supplied it must agree with the information stored on the picture file. The number of fields declared (total number of elements if arrays are

used) must correspond to the total number of fields in the picture.

The reason for the unusual syntax of array declarations can now be explained. Imagine a picture with fields distributed as pictured below

```
A1  B11  C11  B12  C12  D1
A2  B21  C21  B22  C22  D2
A3  B31  C31  B32  C32  D3
A4  B41  C41  B42  C42  D4
A5  B51  C51  B52  C52  D5
```

The fields A1, A2, ... ,A5, B11, B12, ... B52 etc. are fields with the same attributes. This is a situation which often arises when defining pictures to represent tables of information. Also, it is often desirable to be able to access the As, Bs etc. as array elements. The notation for this is A(1), B(1,1), B(1,2), etc. In order to achieve this the following declaration will suffice:

```
FIELD 5(A, 2(B, C), D)
```

In this case the compiler will check each array element against the picture file. The fields may of course be declared as individual fields if desired.

Field names are local to the current picture, which means that while processing a picture one cannot access data from another picture. Also, the same field names may be used on several pictures. Array elements are stored in row major order in the same way as global arrays. The fields are stored in memory in the order in which they appear in the source text. e.g. the declaration

```
FIELD 5(A, 2(B, C), D)
```

will result in the following memory layout:

Elements of A (5 elements)

Elements of B (10 elements)

Elements if C (10 elements)

Elements of D (5 elements)

All data items start on a word boundary. This layout is the record layout which results if a COPY ALL is executed.

The following is an example of a STRUCTURE definition with three pictures.

Example:

```
STRUCTURE FAMILY

1       MALE ID:FIELD ID 1 = 'M'
        FIELD NAME 24, AGE 3.0, PERSON:NUMBER 11.0
        FIELD STREET 20, CITY 15, COUNTRY 10
        FIELD INCOME 8.2

2       FEMALE ID:FIELD ID 1 = 'F'
        FIELD NAME 24, AGE 3.0, PERSON:NUMBER 11.0
        FIELD INCOME 8.2

3       CHILD ID:FIELD ID 1 = 'C'
        FIELD NAME 24, AGE 3.0, PERSON:NUMBER 11.0
        FIELD INCOME 8.2
```

2.5.5 CALCULATION Specifications

The CALCULATION statement has the following format

```
calculation:specification = "CALCULATION" integer .eol
                            e:statements;
```

where integer specifies the picture to which the following calculations apply. The CALCULATION statement is followed by one or more executable statements. These statements are executed when all fields of the current picture are read. The possible executable statements are the following.

```
e:statements = executable:statement $ executable:statement;

executable:statement = compute:statement/

                       accumulate:statement/

                       save:statement/

                       if:statement/

                       loop:statement/

                       display:statement/

                       copy:statement/

                       write:statement/

                       retry:statement/

                       stop:statement;
```

2.5.5.1 Expressions

An expression is built up from operators and operands. An operand may be one of the following:

1. A global variable or array reference.
2. A field or field array reference.
3. A numeric constant.
4. An alphameric constant.
5. An expression enclosed in parentheses.

The operators, listed in order of decreasing priority, are the following

1	*	Multiplication
1	/	Division
1	MOD	Modulo (remainder)
2	+	Addition
2	-	Subtraction
3	space	Concatenation
4	<	Less than
4	<=	Less than or equal to
4	=	Equal to
4	<> or ><	Unequal to
4	>=	Greater than or equal to
4	>	Greater than
5	NOT	Logical negation
6	AND	Logical and
6	OR	Logical or

The operands *, /, MOD, +, and - must have numeric operands and return a numeric result. Minus (-) may also be used as a unary operator. The concatenation operator (space or blank) is valid only with alphameric operands. The result of a concatenation must never exceed 255 bytes in length even if it is for temporary use (e.g. in an IF statement). The relational operators may have both numeric and alphameric operands, but the two operands must be of the same type. The result returned is of type Boolean. such a result cannot be used directly, but only through an IF or a WHILE statement. The operators NOT, AND, and OR takes operands of type Boolean and return a result of the same type. An expression involving these operators is evaluated from left to right until the result (TRUE or FALSE) can be determined, then evaluation ceases and the appropriate action is taken without evaluating the rest of the expression.

All numeric operations are performed on numbers represented in BCD format. If a field is defined to be represented in any other format occurs in an expression, it is converted into BCD in order to perform the required operations. However the field is still stored in the defined format in the data area. All numeric global variables are stored in BCD format.

Array indices which are constants, or constant expressions, are checked at compile time, all other array indices are checked at runtime. An array which is declared as n-dimensional can be accessed by a single index as long as it is within the array. For information about the order of array elements in storage, read the section about declaration of arrays.

The following is a detailed definition of the syntax of expressions.

```
expression =                locical:term $("OR" logical:term);

logical:term =              logical:negation $("AND" logical:negation);

logical:negation =          ["NOT"] relation;

relation =                  concatenation [relational:operator concatenation];

relational:operator =       "<"/ "<="/ "="/ ("<>"/ "><")/ ">="/ ">";

concatenation =             sum $(.blank sum);

sum =                       ["-"] term $(("+"/ "-") term);

term =                      primary $(("*"/ "/"/ "MOD") primary);

primary =                   "(" expression ")"/
                            alpha:constant/ numeric:constant/
                            reference;

reference =                 variable:reference/ field:reference;

variable:reference =        simple:reference/ array:reference
                              <reference to global variable or array>;

field:reference =           simple:reference/ array:reference
                              <reference to field or field array>;

simple:reference =          identifier <previously declared>;

array:reference =           identifier "(" subscript:list ")"
                              <previously declared>;

subscript:list =            expression $("," expression);
```

2.5.5.2 The COMPUTE Statement

The COMPUTE statement which has the format

```
compute:statement = "COMPUTE" variable:reference "FROM" expression .eol;
```

is the assignment statement of the language. The value of the expression to the right is computed and stored in the variable or array element denoted by variable:reference. This must be a global variable, it is not possible to change data read from fields in a picture. The variable reference and the expression must be either both numeric or both alphameric. Alphameric strings are copied left justified and extended with spaces to the right if necessary. No check is performed to ensure that the value will fit in the storage allocated to the variable or that precision will not be lost.

Examples:

```
COMPUTE SUM FROM FIELD1+FIELD2+FIELD9

COMPUTE ARRAY(1,2,J+3) FROM ARRAY(1,1,1)+SUM
```

2.5.5.3 The ACCUMULATE Statement

The ACCUMULATE statement has the following format:

```
accumulate:statement =
       "ACCUMULATE" field:reference "IN" variable:reference .eol;
```

It is semantically equivalent to the statement

```
COMPUTE variable:reference FROM variable:reference+field:reference
```

It provides for a simple way of writing the common operation of accumulating some field in a global variable.

Examples:

```
ACCUMULATE FIELD1 IN SUM

ACCUMULTE INCOME IN TOTAL:INCOME
```

2.5.5.4 The SAVE Statement

The SAVE statement has the following format:

```
save:statement = "SAVE" field:reference "IN" variable:reference .eol;
```

and is semantically equivalent to the statement

```
COMPUTE variable:reference FROM field:reference
```

It provides for an easy way of writing the common operation of saving some field in a global variable for later use.

Example:

```
SAVE FIELD1 IN SAVE:FIELD1
```

2.5.5.5 The IF, ELSEIF, ELSE, and ENDIF Statements

The general form of the conditional statement is:

```
if:statement = "IF" expression .eol
               e:statements
               $("ELSEIF expression .eol
                 e:statements)
               ["ELSE" .eol
                e:statements]
               "ENDIF" .eol;
```

The expressions in the IF and ELSEIF statements must return a result of type Boolean. IF statements can be nested to any desired level. At most one of the groups of statements (e:statements) is executed every time the if statements is executed. If none of the conditions are true and no ELSE clause is provided, no statements are executed. If more than one of the conditions in IF and ELSEIF are true, the statements corresponding to the first true condition are executed. The following examples show some possible forms of the IF statement.

Examples:

```
IF A>B
  COMPUTE TEMP FROM A
  COMPUTE A FROM B
  COMPUTE B FROM TEMP
ENDIF

IF A<B
  ACCUMULATE F1 IN S0
ELSEIF A=B
  ACCUMULATE F1 IN S1
ELSE
  ACCUMULATE F1 IN S2
ENDIF

IF X<=0 OR X>100
  COMPUTE X FROM 199
ELSE
  COMPUTE X FROM 2*X
ENDIF

IF S1><"YES"
  STOP
ENDIF
```

2.5.5.6 THE LOOP, FOR, WHILE, and REPEAT Statements

The general form of a loop is:

```
loop:statement = "LOOP" .eol
                 ["FOR" variable:reference "=" expression
                                              ["TO" expression
                                               ["BY" expression]].eol
                 e:statements]
                 $("WHILE" expression .eol
                   e:statements)
                 "REPEAT" .eol;
```

The LOOP and REPEAT statements delimit the range of the loop. Loops may be nested to any desired level. Exit from the loop may be achieved by means of the WHILE statement. When the expression following WHILE, which must return a result of type Boolean, is no longer true at the time the WHILE statement is executed control is transferred to the statement immediately following the corresponding REPEAT statement. A loop may contain several WHILE statements. The FOR statement causes the loop to be executed as many times as the control expressions specify. If no BY clause is specified BY 1 is assumed. The expressions are evaluated once for every execution of the loop. The loop

```
LOOP
FOR V1=E1 TO E2 BY E3
  .
  .
  .
REPEAT
```

is equivalent to the program

```
    V1:=E1
L1: IF (V1-E2)*SIGN(E3)>0 GOTO L2
      .
      .
      .
    V1:=V1+E3
    GOTO L1
L2:
```

Since the data entry definition language does not contain labels, it is impossible to implements loops in terms of IFs and GOTOs, hence the "escape" to another notation. The above program should however be self explanatory. The function SIGN() returns -1, 0, or 1 depending on whether the argument is negative, zero, or positive.

A loop can contain only one FOR statement and no statements may appear between LOOP and FOR. A loop containing a FOR statement may contain WHILE statements as explained above. For convenience the FOR and WHILE statements can be written on the same line as LOOP. If a WHILE immediately follows a FOR statement it can be written on the same line. The following forms are possible

```
LOOP FOR ....

LOOP WHILE ....

LOOP FOR .... WHILE ....
```

Some examples of possible forms of loops are shown below.

Examples:

```
LOOP FOR I=1 TO 10
  COMPUTE XX(I) FROM XX(I)+1
REPEAT

LOOP WHILE XX(I)><J
  COMPUTE I FROM I+1
REPEAT

LOOP
FOR I=1 TO 40 WHILE OLD(I)<>0
  SAVE OLD(I) IN NEW(I)
REPEAT

LOOP
  COMPUTE I FROM I+1
WHILE TABLE(I)><27
REPEAT
```

2.5.5.7 THE DISPLAY Statement

The DISPLAY statement is used to display messages on the screen during data entry. Messages are displayed right justified on the last line of the terminal. This is the same position that the Data Entry Editor uses for error messages. The length of the message is limited to the length of the screen being used.

During execution of the OUTPUT command in the Data Entry Editor, messages are output to the "error file" together with the work file record number of the current record.

```
display:statement = "DISPLAY" expression .eol;
```

where expression must return an alphameric result.

Examples:

```
DISPLAY "-OK."

DISPLAY "FIELD(3) = " FIELD(3)
```

2.5.5.8 The COPY Statement

The COPY statement is used to copy data from global variables or fields into specified positions in an output record. Two forms exist, one which copies an entire record and another which copies a specified data item.

```
copy:statement = "COPY" ("ALL" "TO" identifier/
                         expression "TO" identifier
                         "(" expression "," expression ")") .eol;
```

The COPY ALL statement copies the whole record as it is received from the screen handling system. The order in which the fields will appear is defined in the section about STRUCTURE definitions. The data is copied to the record specified by the identifier following ALL. The data is copied left justified; i.e. the first field will start in position one of the output record.

If the item to be copied is either an alphameric or numeric value it is copied in byte format into the specified record. The two expressions inside parentheses specify first and last position.

Examples:

```
COPY ALL TO RECORD1

COPY '01' TO RECORD1(1,2)

COPY FIELD(J) TO RECORD(J,J+3)
```

2.5.5.9 The WRITE Statement

The syntax of the WRITE statement is:

```
write:statement = "WRITE" identifier .eol;
```

where the identifier specifies the record to be output. The record name must be previously defined in a RECORD statement. Execution of the statement causes a record of the specified type to be written onto the corresponding file.

Example:

```
WRITE RECORD1
```

2.5.5.10 The RETRY Statement

The RETRY statement causes immediate termination of calulations for the current picture. The value of all global variables are restored to the value they had before calculations started, and the current picture is read once more. The RETRY statement is intended to be used when an error is detected in the input data. The cause of the error may be communicated to the user by immediately preceeding the RETRY statement with an appropriate DISPLAY statement.

```
retry:statement = "RETRY" .eol;
```

2.5.5.11 The STOP Statement

The STOP statement causes immediate termination of calculations for the current picture. The action is the same as if the last statement in this group of executable statements were reached.

```
stop:statement = "STOP" .eol;
```

2.5.6 The END Statement

The END statement signals the compiler that all source lines have been read and compilation is to be terminated. When the compiler listing of the source program is complete, a list of all variables and fields used in the program is output to the list file. Each variable or field is accompanied by a description giving its most important attributes. At the end of the listing the total error count is output. The program is not allowed to run if one or more compilation errors have been reported.

```
end:statement = "END" .eol;
```

2.6 Compiler Error Messages

The following error messages may be output from the compiler in command mode or when entering command mode as a result of a fatal error. Some of these messages are fatal system errors and should never occur if the compiler operates correctly. These messages are indicated by an asterisk.

1. WORKING AREA FULL
2. FILE ERROR
3. * FATAL ERROR
4. * K-STACK UNDERFLOW
5. PICTURE:FILE STATEMENT MISSING
6. STRUCTURE DECLARATION MISSING
7. MISPLACED STATEMENT
8. * BAD PARAMETER PATTERN STRING
9. BAD PARAMETER
10. VALUE OUT OF RANGE
11. ILLEGAL COMMAND
12. AMBIGUOUS
13. STACK OVERFLOW

Diagnostic compiler error messages are output on the compiler listing along with the source lines. Each message is preceeded by three asterisks (***). If no list file is specified, error messages are output to the terminal along with the sequence number of the line in which it was detected. The following is a list of all diagnostic messages.

1. VIOS ERROR nnnn
2. ILLEGAL TERMINATION
3. TOO LONG NAME, TRUNCATED
4. INVALID IDENTIFIER
5. INVALID FILE NAME
6. PICTURE NUMBER USED MORE THAN ONCE
7. SYNTAX ERROR IN ARRAY DECLARATION
8. IDENTIFIER DECLARED MORE THAN ONCE
9. LENGTH INFORMATION MISSING
10. LENGTH EXCEEDS MAXIMUM, ASSUME MAXIMUM
11. DECIMAL POSITIONS EXCEEDS MAXIMUM, ASSUME MAXIMUM
12. INVALID PICTURE NUMBER
13. INVALID RECORD LENGTH
14. IDENTIFIER EXPECTED
15. ID FIELDS MUST APPEAR ON ALL OR NO PICTURE STATEMENTS
16. INVALID FIELD REFERENCE
17. IN, FROM, TO, OR BY EXPECTED
18. INVALID GLOBAL VARIABLE REFERENCE
19. INVALID POSITION SPECIFICATION
20. OUTSIDE RECORD
21. ERROR IN IF NESTING
22. ERROR IN LOOP NESTING
23. IDENTIFIER NOT DECLARED
24. ERROR IN FOR
25. FIELD LENGTH DOES NOT AGREE WITH PICTURE FILE
26. DECIMAL POSITIONS DOES NOT AGREE WITH PICTURE FILE
27. TYPE DOES NOT AGREE WITH PICTURE FILE
28. INCORRECT NUMBER OF FIELDS
29. CALCULATIONS ALREADY SPECIFIED FOR THIS PICTURE NUMBER
30. ILLEGAL EXPRESSION
31. INVALID ARRAY REFERENCE
32. INVALID ARRAY SUBSCRIPT

33. INCORRECT NUMBER OF ARRAY SUBSCRIPTS
34. ILLEGAL OPERATION (TYPE MISMATCH)
35. OUTSIDE ARRAY BOUNDS
36. STRING QUOTE MISSING
37. ALL ID FIELDS MUST HAVE IDENTICAL ATTRIBUTES
38. ERROR IN ID FIELD SPECIFICATION

2.7 Collected Syntax of the Language

```
program =                       picture:file:statement
                                $global:statement
                                $file:and:record:statement
                                structure:definition
                                $calculation:specification
                                end:statement;

picture:file:statement =        "PICTURE:FILE" file:name .eol;

global:statement =              "GLOBAL" declaration:list .eol;

declaration:list =              variable:group $("," variable:group);

variable:group =                variable:definition/
                                dimension "(" variable:group
                                            $("," variable:group) ")";

variable:definition =           identifier .blank length
                                              ["." decimal:positions];

length =                        integer;

decimal:positions =             integer;

file:and:record:statement =     file:statement
                                record:statement $ record:statement;

file:statement =                "FILE" file:name
                                ("DIRECT" record:length/
                                 "sequential"/ "") .eol;

record:statement =              "RECORD" record:list .eol;

record:list =                   record:definition $("," record:definition);

record:definition =             identifier .blank record:length;

record:length =                 integer;

structure:definition =          "STRUCTURE" <arbitrary string> .eol
                                $picture:definition;

picture:definition =            picture:name:statement
                                $field:statement;

picture:name:statement =        integer  identifier
                                ["ID:FIELD" field:definition "=" constant]
                                .eol;

constant =                      alpha:constant/ numeric:constant;

field:statement =               "FIELD" field:declaration:list .eol;

field:declaration:list =        field:group $("," field:group);

field:group =                   field:definition/
```

```
                                dimension "(" field:group $("," field:group) ")";

field:definition =              identifier .blank [length
                                                   ["." decimal:positions]];

calculation:specifications =    "CALCULATION integer .eol
                                e:statements;

e:statements =                  executable:statement $ executable:statement;

executable:statement =          compute:statement/ accumulate:statement/
                                save:statement/ if:statement/ loop:statement/
                                display:statement/ copy:statement/
                                write:statement/ retry:statement/ stop:statement;

compute:statement =             "COMPUTE" variable:reference
                                  "FROM" expression .eol;

accumulate:statement =          "ACCUMULATE" field:reference
                                  "IN" variable:reference .eol;

save:statement =                "SAVE" field:reference
                                  "IN" variable:reference .eol;

if:statement =                  "IF" expression .eol
                                e:statements
                                $("ELSEIF" expression .eol
                                  e:statements)
                                ["ELSE" .eol
                                 e:statements]
                                "ENDIF" .eol;

loop:statement =                "LOOP" .eol
                                ["FOR variable:reference "=" expression
                                          ["TO" expression
                                           ["BY" expression]] .eol
                                 e:statements
                                $("WHILE" expression .eol
                                  e:statements)
                                "REPEAT" .eol;

display:statement =             "DISPLAY" expression .eol;

copy:statement =                "COPY" ("ALL" "TO" identifier/
                                            expression "TO" identifier
                                            "(" expression "," expression ")")
                                .eol;

write:statement =               "WRITE" identifier .eol;

retry:statement =               "RETRY" .eol;

stop:statement =                "STOP" .eol;

end:statement =                 "END" .eol;

expression =                    logical:term $("OR" logical:term);
```

```
logical:term =              logical:negation $("AND" logical:negation);

logical:negation =          ["NOT"] relation;

relation =                  concatenation [relational:operator concatenation];

relational:operator =       "<"/ "<="/ "="/ ("<>"/ "><")/ ">="/ ">";

concatenation =             sum $(.blank sum);

sum =                       ["-"] term $(("+"/ "-") term);

term =                      primary $(("*"/ "/"/ "MOD") primary);

primary =                   "(" expression ")"/
                            alpha:constant/ numeric:constant/
                            reference;

reference =                 variable:reference/ field:reference;

variable:reference =        simple:reference/ array:reference
                              <reference to global variable or array>;

field:reference =           simple:reference/ array:reference
                              <reference to field or field array>;

simple:reference =          integer;

array:reference =           identifier "(" subscript:list ")";

subscript:list =            expression $("," expression ")";
```

3 NORD Data Entry Editor

The NORD Data Entry Editor (DED) is used by an operator to enter data for a specific application. DED replaces the common keypunch machine but is much more versatile.

An operator using a keypunch machine enters data from a form to pre-specified columns on a punched card. These cards are then transferred physically to the host computer where they are read by a card reader.

The Data Entry Editor, on the other hand, presents a picture of the required form on a video display unit and allows the operator to simply copy data from the various fields of the original form to corresponding fields on the video display unit. The operator need not concern himself with the actual column positions. The Data Entry Editor automatically moves to the correct field.

In addition, the operator may go back and modify any field that has been entered incorrectly. In fact, he may at any time go back to previous forms and make changes, delete whole forms or add new forms.

When the operator has finished entering data for an application he may write the data onto any SINTRAN III file for further processing by any subsystem or host computer. A data file produced via a default application may also be read back into the Data Entry Editor such that additional corrections may be made.

3.1 Starting the Data Entry Editor

The NORD Data Entry Editor is started by typing

```
@DED
```

to the SINTRAN III command processor.

DED, upon initialization, identifies itself and then goes into command mode.

3.2 DED Command Processor

The Data Entry Editor command processor uses the bottom line of the video display unit for command input and display of error messages. Commands are entered on the left part of the line while error messages are output right justified.

DED outputs an & whenever it expects a command from the operator. A command consists of a command name followed by zero or more parameters. Several commands, along with all required parameters, may be written on the same line.

The command name consists of one or more parts separated by hyphens ("-"). Each part of the command may be abbreviated as long as the command can be distinguished from all other command names.

While typing commands, the editing characters ctrl-A (backspace one character), ctrl-W (backspace one word) and ctrl-Q (delete whole line) are available.

The collection of parameters is done in a standardized way as follows:

- Parameters are separated by either a comma or any number of spaces or a combination of comma and spaces.
- Parameters may be null in which case a default value is assigned.
- When a parameter is missing, as opposed to null, it is asked for and the command processor expects the operator to supply the required parameter and additional parameters if desired.
- If a parameter syntax error is detected an error message is displayed and the parameter is asked for again.

Control may be returned to the command processor at any time by typing the "escape character". The "escape character" is initially ctrl-G but may be redefined using the SET-PARAMETERS command.

Commands may be given directly to the SINTRAN III command processor by preceeding them with an @ sign. In this case commands to the local command processor following the SINTRAN III command are ignored.

3.3 Function Keys

The Data Entry Editor command processor contains a facility whereby the operator may specify commands with only one key operation. When the operator presses one of these function keys DED will automatically supply the corresponding command. The effect of pressing a funtion key is exactly the same as if the operator types in the complete command name followed by carriage return. Following is a list of the function keys:

ctrl-H	HELP
ctrl-E	EXIT
ctrl-A	APPEND
ctrl-I	INSERT
ctrl-L	LIST
ctrl-N	NEXT
ctrl-P	PREVIOUS
ctrl-D	DELETE
ctrl-C	CHANGE
ctrl-V	VERIFY
ctrl-O	OUTPUT
ctrl-S	SET-PARAMETERS

3.4 Description of DED Commands

Following is a list of the commands interpreted by the Data Entry Editor. Each command is described in detail in the following sections.

- HELP <command name>
- EXIT
- INITIALIZE
- APPEND <picture address>
- INSERT <picture address>
- LIST <picture address>
- NEXT
- PREVIOUS
- DELETE <picture address>
- CHANGE <picture address>
- VERIFY <picture address>
- WRITE-DATA-FILE <file> <format>
- READ-DATA-FILE <file> <format> <picture address>
- OUTPUT <error file> <assignment list>
- SET-PARAMETERS <parameter name> <value>
- MODES

3.4.1 HELP <command name>

The HELP command lists available commands on the display. If <command name> is null, then all available commands are displayed. If <command name> is a legal command name (possibly abbreviated), the command along with its parameters is displayed. If <command name> is ALL, then all commands along with their parameters are displayed.

3.4.2 EXIT

The EXIT command returns control to the SINTRAN III command processor. However, all information relevant to the application is stored on the work file such that the application may be continued at a later time as described in section 3.1.

3.4.3 INITIALIZE

The INITIALIZE command is used to specify which application is to be processed by DED. It may be given at any time and will release any current application before initializing a new one. Any command that requires that an application be present will automatically call the INITIALIZE command.

The INITIALIZE command requests the following information from the operator:

WORK-FILE: <file name> or <carriage return>

The name of a disk file should be specified as the file on which entered data will be stored until it is finally written out onto a data file. If carriage return is typed the system scratch file will be used. The default type for work files is :WORK.

WORK-FILE; OLD OR NEW: <text>

If OLD, or an abbreviation of OLD, is typed then DED will retrieve the remaining parameters from the work file and then enter the normal command mode.

If NEW, or an abbreviation of NEW, is typed then DED will initialize the work file and continue to request the following:

APPLICATION: <file name> or <carriage return>

If the name of an application compiled by the Data Entry Compiler is typed then that application will be read by DED and then the normal command mode will be entered.

If carriage return is typed then DED will initialize the work file as a default application and continue to request the following:

PICTURE-FILE: <file name>

The picture file containing the relevant picture for this application must be entered. The picture file will have been produced by the NORD Screen Picture Maintenance System.

PICTURE-NAME: <name>

The name of the picture (or form) that is required for the current application must be entered. The picture must reside on the picture file specified above.

The application has now been completely specified. All of the parameters concerning the application are stored on the work file such that the data entry process may be interrupted at any time and restarted again by specifying that the work file is OLD.

3.4.4 MODES

The MODES command is used to list the status of various parameters that influence how the Data Entry Editor functions. The following items are displayed with the MODES command:

APPLICATION <application name>

The <application name> is the name of the current application. NONE is displayed if an application has not yet been specified. DEFAULT is displayed if this is a default application.

NO. OF PICTURES <number>

This parameter indicates how many picture records have been entered into the work file. This parameter is not displayed if <application name> is NONE.

CURRENT PICTURE <number>

This parameter indicates which picture record is the current picture. This parameter is not displayed if <application name> is NONE.

TERMINAL-TYPE <terminal name>

This parameter indicates what type of terminal the Data Entry Editor is connected to. The possible <terminal name>s are currently TDV2000, TDV2100, VISTA and INFOTON-200.

TERMINAL-MODE <mode>

This parameter indicates what mode the attached terminal is in. The possible <mode>s are currently ROLL MODE and PAGE MODE.

ESCAPE-CHARACTER <octal value>

This parameter indicates the octal value of the character which is considered to be the "escape character" used when entering data or commands.

READ-STRATEGY <number>

This parameter defines how individual field reading is terminated and how return from a "read fields" call is initiated. Refer to "The NORD Screen Handling System", ND-60.088.01 for more information.

BREAK-STRATEGY <number>

This parameter defines how and when characters typed in on a "read fields" call are to be handled and echoed. Refer to "The NORD Screen Handling System", ND-60.088.01 for more information.

APPEND <mode list>
CHANGE <mode list>
VERIFY <mode list>

The <mode list> specifies whether or not the functions specified therein are to be performed for their corresponding command.

CLBUF=1 indicates that the data record is to be cleared to nulls or spaces before a "read fields" call.

CFLDS=1 indicates that the fields on the display unit are to be cleared before a "read fields" call.

SREAD=1 indicates that all "field read" bits are to be set before a "read fields" call. This enables the operator to copy "old fields" using the appropriate edit functions.

SMUST=1 indicates that all "must read" bits are to be set before a "read fields" call. This forces the operator to enter all fields on the picture.

Refer to "The NORD Screen Handling System", ND-60.088.01 for more information concerning these functions.

3.4.5 SET-PARAMETERS <parameter name> <value> ...

The SET-PARAMETERS command is used to change the value of the various parameters described above for the MODES command.

If carriage return is entered instead of <parameter name> then a list of all the possible <parameter name>s along with their possible <value>s will be shown on the display unit. Several parameters may be changed at once simply by adding more <parameter name> <value> groups to the command line.

3.4.6 APPEND <picture address>

The APPEND command is used to enter data into the work file. The optional <picture address> specifies at what point in the work file that the data should be entered. If <picture address> is null then data is entered at the end of all other entered data.

The <picture address> may be one of the following:

<number> Specifies the <number>th data record in the work file.

$ Specifies the last data record in the work file.

$-<number> Specifies the <number>th data record preceding the last data record.

. Specifies the current data record, i.e., the one just APPENDed, LISTed, CHANGed, etc.

.+<number> Specifies the current+<number>th data record.

.-<number> Specifies the current-<number>th data record.

For those familiar with the text editor QED it can be mentioned that the <picture address> is exactly analogous to the <line address> in QED. In fact, all of the commands in DED are as similar, in function, to those of QED, as possible.

When the APPEND command is given, the Data Entry Editor will display the appropriate picture (form) on the video display unit and expect the operator to fill in all of the variable fields. The way in which this is done is described in "The NORD Screen Handling System", ND-60.088.01.

If the current application uses identification fields, then the first field that is entered is the identification field. If the value entered into the identification field identifies the currently displayed picture, then registration may continue immediately. If the identification field specifies another picture type, then that picture will be displayed before registration can continue.

If the current application does not use identification fields and the entry of data is terminated by ctrl-L, then the data record read from the picture will be added to the work file and thereafter the next picture type will be displayed. If the current picture type is the last picture type then registration will continue with the first picture type of the application.

If the "escape character" is typed, DED will immediately terminate reading of the picture and will return directly to the command processor without adding the data record to the work file. If any other terminating character is typed the data record will be added to the work file and another picture record may be entered.

The APPEND command requires that an application be present, and will therefore automatically call the INITIALIZE command if necessary.

3.4.7 INSERT <picture address>

The INSERT command functions exactly as APPEND except that new data records are entered immediately ahead of the addressed picture record.

The INSERT command requires that an application be present, and will therefore automatically call the INITIALIZE command if necessary.

3.4.8 LIST <picture address>

The LIST command is used to display a data record along with its corresponding picture (form). The optional <picture address> specifies which data record is to be displayed. If <picture address> is null then the current data record is shown on the video display unit. The data record that is displayed becomes the new current data record. The picture address of the displayed data record is displayed right justified on the last line of the display unit.

The LIST command requires that an application be present, and will therefore automatically call the INITIALIZE command if necessary.

3.4.9 NEXT

The NEXT command is used to display the data record following the current data record. The NEXT command is equivalent to LIST .+1. The data record that is displayed becomes the new current data record. If the current data record is the last one in the work file then the errror message END OF WORK FILE will be displayed.

The NEXT command requires that an application be present, and will therefore automatically call the INITIALIZE command if necessary.

3.4.10 PREVIOUS

The PREVIOUS command is used to display the data record preceeding the current data record. The PREVIOUS command is equivalent to LIST .-1. The data record that is displayed becomes the new current data record. If the current data record is the first one in the work file then the errror message BEGINNING OF WORK FILE will be displayed.

The PREVIOUS command requires that an application be present, and will therefore automatically call the INITIALIZE command if necessary.

3.4.11 DELETE <picture address>

The DELETE command is used to delete a data record from the work file. The <picture address> specifies which data record is to be deleted. If <picture address> is null then the current data record is deleted. The record preceeding the deleted data record becomes the new current data record. If the work file is empty the error message NOTHING IN WORK FILE is displayed.

The DELETE command requires that an application be present, and will therefore automatically call the INITIALIZE command if necessary.

3.4.12 CHANGE <picture address>

The CHANGE command is used for modifying fields in the data record specified by <picture address> or the current data record if <picture address> is null.

The specified data record is displayed on the video display unit as with the LIST command. Thereafter the operator may position the cursor to any variable field and change it. The manual "The NORD Screen Handling System", ND-60.088.01, describes how variable fields may be edited.

When the operator has terminated editing, the new data record replaces the old one in the work file except if editing was terminated by the "escape character" in which case nothing is changed.

The CHANGE command requires that an application be present, and will therefore automatically call the INITIALIZE command if necessary.

3.4.13 VERIFY <picture address>

The VERIFY command is used when the operator wishes to control that previously input data records were input correctly. The <picture address> specifies at what point in the work file control reading is to start. If <picture address> is null then control reading starts with the current data record.

When the VERIFY command is specified, the picture (form) corresponding to the specified data record is displayed on the video display unit. The variable fields from the data record are not displayed. The operator is then required to input the fields anew such that they may be compared to the originally input fields. If a discrepancy occurs, the operator is informed by a beep. The operator may then choose whether to keep the old field value or use the new field value. This "control reading" procedure is described in detail in the manual "The NORD Screen Handling System", ND-60.088.01. Any new information resulting from the VERIFY operation replaces the old information in the work file.

After VERIFYing the first specified data record the operator is requested to "control read" the following data records until termination via the "escape character" occurs, as in the APPEND command.

The VERIFY command requires that an application be present, and will therefore automatically call the INITIALIZE command if necessary.

3.4.14 WRITE-DATA-FILE <file> <format>

The WRITE-DATA-FILE command is used to output all of the data records in the work file to the operator-specified <file>. The type of output file is specified by <format> as follows:

DIRECT — Each data record is written onto the output file as a fixed size block. The block size is determined by the size and format of the individual fields as specified in the manual "The NORD Screen Handling System", ND-60.088.01. The output file is terminated with a block containing an end of file character (27 octal) in the first byte position. The default file type is :DATA.

SEQUENTIAL — This file format should only be used for ASCII data. Spaces are removed from the end of each data record and carriage return line feed is added such that this type of file will be compatible with SINTRAN III symbolic files. The file is terminated with an end of file character. The default file type is :SYMB.

The WRITE-DATA-FILE command is applicable only to default applications, and will therefore automatically call the INITIALIZE command if necessary.

3.4.15 READ-DATA-FILE <file> <format> <picture address>

The READ-DATA-FILE command may be used to read data records from the specified file into the work file at the specified <picture address>. Data records from the input file are in effect APPENDed after the data record specified by <picture address>. If <picture address> is null then records are APPENDed after the last data record in the work file.

The <format> must correspond to the format used to create the input file with the WRITE-DATA-FILE command.

The READ-DATA-FILE command is applicable only to default applications, and will therefore automatically call the INITIALIZE command if necessary.

3.4.16 OUTPUT <error file> <assignment list>

The OUTPUT command is used to output the final data records specified for the current application.

The <error file>, which may be null, will contain all error messages generated during the output process. These error messages may come from either the Data Entry Editor or user supplied DISPLAY statements.

The <assignment list> specifies the mapping of logical to physical files and also whether data is to be appended to the physical file or replace its previous contents.

For each logical file in the application, DED will ask for the following, if not already supplied on the command line:

ASSIGN <logical file> TO <physical file>

<logical file> is supplied by DED.
<physical file> must be supplied by the operator.

APPEND OR REPLACE: <text>

The operator must supply <text> which should be APPEND, REPLACE or an abbreviation thereof.

The OUTPUT command requires that an application be present, and will therefore automatically call the INITIALIZE command if necessary.

3.5 DED Error Messages

All error messages from the Data Entry Editor are displayed right justified on the last line of the display unit. Following is a list of all error messages that may occur:

1. FILE ERRROR
2. WORKING AREA FULL
3. NOTHING IN WORK FILE
4. END OF WORK FILE
5. BEGINNING OF WORK FILE
6. ILLEGAL PICTURE ADDRESS
7. NOT AN APPLICATION FILE
8. COMPILATION ERROR IN APPLICATION
9. VALID ONLY FOR DEFAULT APPLICATION
10. NO FILE SUPPLIED
11. NOT A WORK FILE
12. BAD PARAMETER
13. VALUE OUT OF RANGE
14. NOT FOUND
15. AMBIGUOUS
16. STACK OVERFLOW
17. ARRAY INDEX OUT OF RANGE
18. ATTEMPT TO DIVIDE BY ZERO
19. ATTEMPT TO WRITE OUTSIDE RECORD
20. PICTURE NOT IDENTIFIED

NORSK DATA A.S.
Lørenveien 57 - Postboks 163, Økern
OSLO 1

COMMENT AND EVALUATION SHEET

Publication No. ND-60.101.02 Nord Data Entry System
December 1977 Reference Manual

In order for this manual to develop to the point where it best suits your needs, we must have your comments, corrections, suggestions for additions, etc. Please write down your comments on this pre-addressed form and post it. Please be specific wherever possible.

FROM _____

_____

_____