HONEYWELL

MULTICS REPORT PROGRAM GENERATOR (MRPG) REFERENCE MANUAL

SOFTWARE

SERIES 60 (LEVEL 68)

MULTICS REPORT PROGRAM GENERATOR (MRPG) REFERENCE MANUAL PRELIMINARY EDITION

SUBJECT

Detailed Description of the Multics Report Program Generator (MRPG) Including Details of the Language Necessary to Prepare, Generate, and Execute Programs

SPECIAL INSTRUCTIONS

This manual presupposes some basic knowledge of the Multics system, and does not attempt to provide extensive information on a text editor needed to write MRPG programs, on the PL/I compiler, nor on the methods that may be used to create and maintain the input files from which reports are produced.

This preliminary edition is based on the best information available at the time of writing. Additional functional capabilities are expected to be provided in future revisions.

SOFTWARE SUPPORTED

Multics Software Release 6.0

ORDER NUMBER

CC69, Rev. 0

March 1978

Honeywell

PREFACE

This reference manual contains a complete definition of the Multics Report Program Generator (MRPG) language. Programmers can use this language to write MRPG programs that produce simple or complex formatted reports. Numerous examples illustrate every portion of the MRPG language. Some of the examples clarify interactions between MRPG programs and other Multics procedures.

The reader is assumed to:

- Know how to log on
- Know how to use a text editor
- Know how to invoke object segments as Multics commands
- Know how to provide arguments with commands
- Have a general knowledge of the virtual memory
- Have a general knowledge of I/O switches and attachments

Throughout this manual, references are frequently made to other Multics manuals. For convenience, these references are as follows, where MPM stands for Multics Programmers' Manual:

DOCUMENT	REFERRED TO IN THE TEXT AS
MPM Reference Guide (Order No. AG91)	MPM Reference Guide
MPM Commands and Active Functions (Order No, AG92)	MPM Commands
MPM Subroutines (Order No. AG93)	MPM Subroutines
MPM Subsystem Writers' Guide (Order No. AK92)	MPM Subsystem Writers' Guide
MPM Peripheral Input/Output (Order No. AX49)	MPM I/O
Multics PL/I Reference Manual (Order No. AM83)	Multics PL/I Manuals
Multics PL/I Language Specification (Order No. AG94)	Multics PL/I Manuals

CONTENTS

ILLUSTRATIONS

TABLES

SECTION 1

INTRODUCTION

The Multics Report Program Generator (MRPG) is a language translator used to generate a PL/I source program from an MRPG source program with the purpose of generating formatted reports. A complete definition of the language is presented in a COBOL-like notation (see Section 5).

SUMMARY OF THE MRPG APPROACH

The source program may be built with a text editor in a free-form format using the MRPG language that is much higher in level than procedural languages such as BASIC, FORTRAN, COBOL, or PL/I. A PL/I source program is generated from the MRPG source program. The standard PL/I compiler then compiles the PL/I source program into an MRPG object segment (MRPG-OS). An ASCII input file is read by the object segment and one or more reports are produced. A report can be printed on the user's terminal as it is being produced or it may be written to a segment for later printing.

Once an object segment has been created, it can be used repeatedly with input files that have the same structure.

Conditional tests may be used to decide whether to include or omit lines and/or fields in reports. Control breaks on input fields can produce detail summary lines. Report and page heading and footing capabilities are provided.

It is assumed that the person writing source programs is an experienced programmer. However, the person triggering the running of an object program need not be a programmer, whether such triggering is done from Multics command level or by a Logical Inquiry and Update System (LINUS) report request.

COMMAND PROCESSOR VS. I/O MODULE

An MRPG-OS can be invoked from Multics command level in exactly the same manner as system commands, for example:

```
name_of_the_MRPG-OS {arguments}
```

When using this method, the input file must be in an existing segment before the MRPG-OS is invoked.

Another method, the one used by LINUS, involves the report_ I/O module. ASCII records built by another program are sent via report_ to the MRPG-OS.

MULTIPLE USE OF INPUT DATA

In the simple, straightforward case, the input records are processed as they are received to produce the output report. An MRPG program can process the input data more than once. For instance, produce a report from the input data records in their original sequence, sort them into another sequence and produce a different report. When such multiple passes over the input data occur, the original input is read only once and is saved in a temporary file for later reuse.

EXAMPLES

Section 2 presents a complete example that includes examining the data available in a file, writing the source program, executing the object segment, and the printing of reports. Other sections contain fragments of programs, where the fragments are chosen to illustrate specific points.

As an aid to the person becoming familiar with MRPG, the input file and the source program described in Section 2 are also available on the system (if the unbundled MRPG has been ordered and received). The reader can copy and modify the source program, generate a new object program, and run it using the same input file as is used in Section 2. A few more example programs are also provided on the system with the MRPG package. See Appendix B for the details concerning the content and location of the example input files and source programs.

SECTION 2

A COMPLETE EXAMPLE

The example described in this section is intended to assist the reader in becoming familiar with MRPG. This is an artificial example designed to illustrate many MRPG features, and therefore, may appear to be moderately complicated when it is read for the first time. Section 5 contains a large number of examples of individual statements, each of which is treated in an isolated manner. This section provides an integrated, complete example.

An input file, an output report, and the MRPG source program used to produce the report are shown. Within the source program, the "||" symbol stands for the concatenation operation. The ":=" symbol specifies an assignment operation.

The input file and the MRPG source program are available on the user's system if the unbundled software is installed. The MRPG input file and the source program are in an archive:

```
Archive:          >system_library_unbundled>mrpg_examples.archive
Input File:       filing_cabinet.mrpg.input
Source Program:   filing_cabinet.mrpg
```

Appendix B explains how to obtain and run this example.

A discussion of the actions that are involved in building the input, preparing the source program, converting the source program into an object program, and producing the report are included with the example.

THE INPUT FILE

Figure 2-1 contains a nine record input file. The file is a segment containing only ASCII characters. Each record ends with a newline character. Neither the heading lines nor the left column (Record) are present in the input file. They are included only for purposes of illustration and ease of understanding. The four data columns and the remarks column comprise the file. The remarks are ignored because the file is declared as a stream file and only the first four fields are declared as being part of the file.

THE REPORT

Figure 2-2 is the report produced when the program in Figure 2-3 receives the input file shown in Figure 2-1 and no arguments are supplied at the time that the object program is invoked. The column of line numbers at the left of the report are not part of the report, but are included to simplify discussing specific lines in the report. If the -file argument was supplied when the object program was invoked, then lines 1 to 3 of Figure 2-2 would not be produced and a newpage character (014 octal) would follow the last line. The report contains only ASCII characters.

```
Record
No.                          The Actual Input Data Records
----  ------------------------------------------------------------------------

 1    u  2   10   30  The input data lines are sorted in the desired
 2    u  4   34   50  order.  Therefore, another sort phase is not required in
 3    u  5    1   60  the program.
 4    c  2   32   50  These comments are ignored by the procedures that read the
 5    c  4  100   84  input (i.e., read the 14 declared characters and skip until
 6    c  5   40  100  the next newline character is passed over).  Skipping
 7    f  2    0  100  to a newline occurs because the input file declarations
 8    f  4    8  250  include the "stream" keyword.  Each record in this file
 9    f  5    3  300  ends with a newline character.
```

Figure 2-1. A Sample MRPG Input File

```
Line
No.                          The Actual Output Lines
----  ------------------------------------------------------------------------

 1
 2
 3
 4    FILING CABINET INVENTORY AS OF 02/13/78
 5
 6
 7    Grade:  Commercial --  Purchased from:  Cranston Office Furniture
 8
 9              No. of Drawers Quantity  Unit Cost  Extended Cost
10
11    ----+----1----+----2----+----3----+----4----+----5----+----6
12
13                   2             32   $   50        $ 1,600
14                   4            100   $   84        $ 8,400
15                   5             40   $  100        $ 4,000
16
17       TOTALS: QUANTITY =        172          COST  $ 14,000
18
19
20    Grade:  Fireproof --  Purchased from:  Firesafe Specialities
21
22              No. of Drawers Quantity  Unit Cost  Extended Cost
23
24    ----+----1----+----2----+----3----+----4----+----5----+----6
25
26                   2              0   $  100        $     0
27                   4              8   $  250        $ 2,000
28                   5              3   $  300        $   900
29
30       TOTALS: QUANTITY =         11          COST  $  2,900
31
32
33    Grade:  Utility --  Purchased from:  Universal Metal Products
34
35              No. of Drawers Quantity  Unit Cost  Extended Cost
36
37    ----+----1----+----2----+----3----+----4----+----5----+----6
38
39                   2             10   $   30        $   300
40                   4             34   $   50        $ 1,700
41                   5              1   $   60        $    60
42
43       TOTALS: QUANTITY =         45          COST  $  2,060
44
45
46       GRAND TOTALS: QUANTITY =   228       COST    = $  18,960
```

Figure 2-2. A Sample MRPG Report

```
Line
No.                          The Actual Source Program Lines
----  ------------------------------------------------------------------------

/* A sample program to illustrate many of the MRPG features.  This is
 *       filing_cabinet.mrpg  in  >unbundled>mrpg_examples.archive  */

declare 1 parameter, 2 where_to_send_output boolean key ("-file");

declare 1 input  stream
        file "filing_cabinet.mrpg.input",
        2 grade              char(1)  position  1,
        2 drawers            dec(1)   position  4,
        2 quantity           dec(3)   position  7,
        2 unit_cost          dec(3)   position 12;

declare quantity_total  dec;   declare quantity_grand_total  dec;
declare cost_total      dec;   declare cost_grand_total      dec;

declare grade_code_to_name table
  ("c" -> "Commercial"  "f" -> "Fireproof"  "u" -> "Utility") varying;
declare grade_code_to_supplier table
  ( "c" -> "Cranston Office Furniture"
    "f" -> "Firesafe Specialities"
    "u" -> "Universal Metal Products" ) varying;

define 1 report filing_cabinet_inventory   break (grade)   pagelength 46
       on (file "filing_cabinet.report" if (where_to_send_output)
           or switch "user_output"),
2 pagehead, 3 line 4, 4 "FILING CABINET INVENTORY AS OF "  ||  %mmddyy,
            3 line +2,
2 detailhead grade,
 3 line +3,
  4 "Grade:  ",
  4 transform (grade, grade_code_to_name)  let (quantity_total := 0;),
  4 "--  Purchased from:  ",
  4 transform (grade, grade_code_to_supplier) let (cost_total := 0;),

 3 line +2,
  4 "No. of Drawers"     column 11,     4 "Quantity"          column 26,
  4 "Unit Cost"          column 36,     4 "Extended Cost"     column 47,
 3 line +2,
/* Next line provides column numbers for the reader's convenience. */
  4 "----+----1----+----2----+----3----+----4----+----5----+----6",
 3 line,
2 detail the_data_line,
 3 line,
  4 drawers                    column 17 picture "9",
  4 quantity                   column 31 picture "zz9",
  4 unit_cost                  column 37 picture "$z,zz9",
  4 quantity * unit_cost       column 51 picture "$zz,zz9"
    let ( quantity_total := quantity_total + quantity;
          cost_total := cost_total + quantity * unit_cost; ),
2 detailfoot grade,
 3 line +2,
  4 "   TOTALS: QUANTITY = ",
  4 quantity_total             column 30 picture "zzz9",
  4 "COST"                     column 44,
  4 cost_total                 column 50 picture "$zzz,zz9"
   let (quantity_grand_total := quantity_grand_total + quantity_total;
        cost_grand_total := cost_grand_total + cost_total; ),
2 pagefoot,
 3 line 46,
  4 "   GRAND TOTALS: QUANTITY = ",
  4 quantity_grand_total       column 29 picture "zzzz9",
  4 "COST   ="                 column 40,
  4 cost_grand_total           column 49 picture "$zzzz,zz9";

begin ( quantity_grand_total := 0; cost_grand_total := 0;)
-print the_data_line;    end;
```

Figure 2-3. A Sample MRPG Source Program

THE SOURCE PROGRAM

Figure 2-3 shows the MRPG source program. A text editor is used to build the lines of the source program and to write the program into a segment with the suffix mrpg as the last component of the segment's name. The program could have been built with fewer characters and with fewer lines. It was deliberately built as it is to clearly show the hierarchal structure of the input file and of the report definition. The source program contains only ASCII characters.

THE OBJECT PROGRAM

The source program in Figure 2-3 is converted into a standard Multics object segment as follows:

1. Change to a directory in which the user's process can create segments.

2. Extract an exec_com from the archive by typing:

```
ac x >unb>mrpg_examples run_mrpg_examples.ec
```

3. Invoke the exec_com by typing:

```
ec run_mrpg_examples
```

4. A menu of sample programs is displayed. A request is made to type in one of the menu numbers. Type in the menu number for the filing_cabinet example.

5. The MRPG language translator types out MRPG, reads in and processes the source program, and generates a PL/I source program that is written into the user's working directory with the name:

```
filing_cabinet.pl1
```

   The MRPG then automatically invokes the PL/I compiler. It types out PL/I and compiles the PL/I source program into an object program in the user's working directory with the name:

```
filing_cabinet
```

6. The user is asked if any more examples are to be run. Type "no" and control will return to the Multics command level. If ready messages are enabled, a ready message is typed.

PRODUCING THE REPORT

A choice must be made, namely, where is the report to appear. If it is to be typed out directly onto the terminal, then step 1 below is applicable. If the report is to be written into a segment for later use, then step 2 is relevant.

1. Invoke the object program by typing:

```
filing_cabinet
```

   The object program reads in the input file and types out the report on the user's terminal after which control returns to command level.

2. Invoke the object program by typing:

```
filing_cabinet -file
```

The object program reads the input file and writes the report into a segment named:

```
filing_cabinet.report
```

in the user's working directory after which control returns to command level. The following paragraph discusses how to have the segment printed.

PRINTING THE REPORT

A report that is in a segment in the user's working directory may be displayed on a terminal by typing:

```
print filing_cabinet.report
```

or may be printed on the high-speed line printer by typing:

```
dprint filing_cabinet.report
```

SECTION 3

LANGUAGE CONCEPTS

The MRPG language is defined in Section 5 with some introductory material and 18 sets of information. The term "group" is used to refer to one of these 18 sets. Each group contains a COBOL-like general format diagram, and (1) examples, (2) syntax rules, and (3) general rules applicable to the group.

RELATIONSHIP TO PL/I

Because the MRPG language translator generates a PL/I source program that is then compiled by the standard PL/I compiler, some MRPG language characteristics are actually PL/I characteristics. An example is the current limit of 256 characters for the length of a user-defined name. Another example is the definition and treatment of pictures ("zzz,zz9v.99"). Should such characteristics change in PL/I, then the new PL/I characteristic also becomes -- instantaneously -- the new MRPG characteristic.

For complicated characteristics, rather than duplicate substantial portions of the PL/I manual in this manual, references are made to the PL/I manuals. In some instances, the amount of text required to state the PL/I characteristic is small. In these cases, the PL/I characteristic is restated in this manual. Whether or not the PL/I characteristic is stated in this manual, or merely a reference appears, this manual identifies such characteristics with the following sentence:

This is a PL/I characteristic.

The significance of the above remark is that the authoritative, governing documentation is found in the PL/I manuals.

CHARACTER SET

The entire 7-bit ASCII character set is available for use in an MRPG source program and in the input file.

An MRPG source program can be thought of as containing three domains:

- Quoted strings ("A quoted string.")
- Comments (/* A comment. */)
- Everything else

The full ASCII character set can be used for quoted strings, comments, and in the input file without causing problems.

Only the following characters are valid in the "everything else" domain. When nonprinting characters that are not listed below are encountered between language elements, such nonprinting characters are ignored (normal program generation and compilation occur). Nonvalid printing characters encountered between language elements cause an MRPG warning message to be sent to the user and then these nonvalid printing characters are ignored (normal program generation and compilation occur). Nonvalid characters within language elements yield error messages and program generation does not occur.

Valid printing characters:

The 52 uppercase and lowercase letters.
The 10 digits.
, Comma
; Semicolon
(Left parenthesis
) Right parenthesis
_ Underline (Underscore)
+ Plus
- Minus (Hyphen, Dash)
* Asterisk
/ Right slant (Slash)
< Less than
= Equals
> Greater than
& Ampersand
¦ Vertical line (Vertical bar)
^ Circumflex (Caret)
. Period (Decimal point, Dot)
% Percent
: Colon
" Double quote (Quotation mark, Quote)

Valid nonprinting characters (white space):

Space
Horizontal tab
Newline (Line feed)
Vertical tab
Newpage (Form feed)

Several of the printing characters are valid only at particular places in an MRPG source program. Examples are the & (for a logical AND operation) and the % (to call a builtin function). These cases are described in Section 5.

INPUT FILE

The input file is assumed to be a file of 9-bit ASCII characters. Any ASCII character may reside in the lower seven bits of each 9-bit character. The two high-order bits of each 9-bit character must be zeros.

The file must be a canonical file. See the MPM Reference Guide for the definition of and a discussion about canonical files. The "canon" command, described in the MPM Commands, may be used to convert a non-canonical file into a canonical file.

CHARACTER COMBINATIONS

The < > character combinations are not part of the MRPG language. They are used to form abbreviations in this section and include:

<SP> space

<NL> newline (line feed)

<HT> horizontal tab

<VT> vertical tab

<NP> newpage (form feed)

<BSP> backspace

Conventional keywords are formed from printable characters. Special meanings attached to certain printable character combinations are:

|| concatenate operator

:= assignment operator

/* start of a comment

*/ end of a comment

-> translation operator (in tables; see the Declare_Variable group)

FI a keyword denoting the end of an IF statement in the Execute_Phrase group

... conventional ellipsis

BSP a keyword specifying that backspace characters may be present in the character expression (see the Report_Field_Def group).

ELEMENTS OF THE LANGUAGE

A source program contains the following types of elements:

- MRPG reserved words
- Comments
- Separators
- User-defined names
- Quoted strings
- Integers and numbers

These words may be used in MRPG source programs, but must not appear in the program as user-defined names. A complete list is given in Appendix A.

KEYWORDS

A keyword is an MRPG reserved word whose presence is required when the format in which the word appears is used in a source program. A few of the keywords are:

```
DECLARE
PAGELENGTH
PRINT
VAR
VARYING
```

Unlike COBOL, MRPG has no optional words.

Within each format, the keywords are shown in uppercase. However, when they are used in a source program, they can be in lowercase, uppercase, or a mixture. That is, declare, DECLARE, Declare, and dEcLaRe are equivalent.

In one sense, the digits 0, 1, 2, 3, and 4 are keywords, because they are specifically called for in certain formats, for example:

```
DEFINE  1  REPORT  ...,
        2  REPORTHEAD  ...
```

but these digits are also available for use elsewhere in the source program. Therefore, these digits are not reserved words.

PUNCTUATION

Only the comma and the semicolon are used as punctuation characters. Both the comma and the semicolon are used to delimit major portions of the program. In such usage, the comma is at a lower hierarchial level than the semicolon. In addition, the comma may be used to separate items in a list. Section 5 specifies where commas and semicolons must be used.

SPECIAL-CHARACTER WORDS

- Arithmetic operators (e.g., + and -;)
- Relational operators (e.g., < and =)
- Logical operators (e.g., & and ^)

may be thought of as reserved words, as is done in COBOL, because they have special meanings within the MRPG language and because they are not available for indiscriminate use by the programmer.

Comments

Comments can only exist outside of quoted strings and between language elements.

A comment begins with a /* (character pair) and ends with the next */ (character pair). The beginning and ending /* and */ characters are considered part of the comment.

The /* */ pairs and all intervening characters are ignored by the MRPG language translator.

The following example illustrates several situations of interest. In each case shown below, the English text between the /* and the next */ is a true remark about the situation. Notice that one of these situations is not valid.

```
/* A comment can be at the beginning of the segment that contains the MRPG
   source program. */

declare  1  parameter  ...
   .
   .
   .
decla/* This will result in an error. */re  1  ...
   .
   .
   .
declare/* Valid to have the starting /  immediately after the end of a word
          and also  valid for the  ending / to  immediately  precede a word
          (i.e., it is not  necessary for a space  character to precede the
          /* pair nor to follow the */ pair).  */1 input ...
   .
   .
   .

/* This six-line comment includes four blank lines.



*/

declare average decimal; /* Explanatory remarks. */
   .
   .
   .

define 1 report payroll

/* If the * / are separated, they do not end the comment. */

pagewidth 132

/* Another /* does not  "nest" comments.  Only  one asterisk right slant is
   needed to terminate a comment. */

pagelength 66
   .
   .
   .

/* The next two lines divide the sum by a count. */
average_1 :=  sum/* Confusing, but valid comment.*// count_1;
average_2 :=  sum//* This is also valid. */count_2;
```

```
    .
    .
end;

/* A comment can be at the end of the segment that contains the MRPG source
   program. */
```

Separators

Elements of an MRPG source program are separated by one or more separator characters. The most commonly used separator characters are the white space characters. Their name, octal value, and graphic representation in this section are:

Space	octal 040	<SP>
Horizontal tab	octal 011	<HT>
Newline (Line feed)	octal 012	<NL>

Other white space characters are:

Vertical tab	octal 013	<VT>
Newpage (Form feed)	octal 014	<NP>

In a strict technical sense, the characters discussed in the next few paragraphs are not separators. However, they can perform the function of delimiting language elements.

When called for in a general format, the following characters separate and delimit language elements:

Comma	,
Semicolon	;
Colon followed by equals	:=
Quote	"
Left parenthesis	(
Right parenthesis	)
Hyphen followed by greater than	->
Percent	%

In addition, the /* and */ (character pairs), which delimit the start and end of comments, also separate and delimit language elements.

The nonalphabetic language elements:

```
+  -  *  /  ,  ;  <  <=  =  >=  >  ^=  ¦  &  ^  ¦¦  (  )  "  :=  ->
```

also function as separators, and therefore, do not need to be surrounded by separators, but their alphabetic equivalents, such as LT and NOT, must be surrounded by separators.

In these examples, when there is a gap between characters, that gap represents one or more white space characters.

1. The line:

```
dcl <SP>1<SP>input
```

is equivalent to:

```
dcl<HT><HT>1<HT><SP><HT>input
```

2. The line:

```
dcl fives set (5, 10, 15) ;<NL>
```

is equivalent to:

```
dcl fives set(5,10,15);<NL>
```

which is equivalent to:

```
dcl<SP>fives<NL>
<NL>
<NL>
<NL>
set(5,<NL>
10,15);<NL>
```

3. The line:

```
dcl kind table ( 2 -> "Bicycle" 4 -> "Car" ) ;<NL>
```

is equivalent to:

```
dcl kind table (2->"Bicycle"4->"Car");<NL>
```

4. The line:

```
dcl subtotal dec ;<NL>
dcl total dec ;<NL>
```

is equivalent to:

```
dcl subtotal dec;dcl total dec;<NL>
```

5. However, the quoted string:

```
"United<SP>States"
```

is different than:

```
"United<HT>States"
```

and both differ from:

```
"United<NL>
States"
```

although all three forms are valid quoted strings.

User-Defined Names

A user-defined name is an MRPG word that the user specifies to satisfy the format of a clause or statement. The characters that may be used are the 52 uppercase and lowercase letters, 10 decimal digits (0-9), and the underscore character (_). All of the names defined by the user must be unique and must begin with one of the 52 alphabetic characters. The maximum length of a user-defined name is 256 characters. These are PL/I characteristics.

In all the groups, the lowercase character strings that have a _name suffix identify those places where the user must define a name. The complete list of these names is:

```
parameter_name
input_field_name
local_variable_name
set_variable_name
table_variable_name
report_name
detail_name
```

The PL/I source program that is generated and compiled based on the user's MRPG source program uses several internal names. The internal names all begin with a two-character sequence of one uppercase letter followed by one underscore character. Therefore (to avoid conflict), no user-defined name can begin with an uppercase letter followed by an underscore.

Unlike the situation with MRPG keywords where uppercase letters are not distinguished from lowercase letters (e.g., declare and dEcLaRe are equivalent) the distinction is made for user-defined names. Thus, Payroll and payroll are two unique names.

Quoted Strings

The reader will encounter elements such as "string-1" at several places throughout the group diagrams in Section 5. Any ASCII character can be placed between the starting quote character and the ending quote character.

If the user intends to construct a printed string that contains a quote in the output, two adjacent quotes for each quote desired in the output plus one additional quote at both the beginning and end of the string must be supplied.

MRPG Source Program	Printed Output
"The simple case."	The simple case.
"""Normal quoting."""	"Normal quoting."
"""""Double quotes."""""	""Double quotes.""
"Quote in "" middle."	Quote in " middle.

The maximum length of a quoted string in the source program is 254 characters. This count is determined after:

1. Removing the string containing quotes (beginning and end)
2. Replacing all pairs of adjacent quote character with one quote character

This is a PL/I characteristic.

Integers and Numbers

Integers can be formed from the ten decimal digits (0-9). Internally, integers are kept in fixed binary(35) format.

Numbers can be formed from the ten decimal digits (0-9) and the decimal point. Internally, numbers are kept in float decimal(20) format. The maximum quantity of significant digits that are retained to express the value of a number is 20 digits. (Refer to "Table 6-2" for an example of a 20-digit number.)

DEFAULTS

If certain phrases are omitted in a program, default actions are taken, or default values are assumed. (Refer to Section 5 for specific default conditions related to the individual group description.)

DEFINE BEFORE REFERENCE

A data item must be defined in a DECLARE OR DEFINE statement before a reference can be made to that data item. The definition can occur in an earlier portion of the statement in which the reference occurs. Some examples are:

```
declare 1 input,  /* this example is valid */
        2 length_of_name dec(4),
        2 item_name char(length_of_name),
        2 item_quantity dec(6);
```

```
declare 1 input,  /* but this example is in error, because the reference to
                     length_of_name in the parentheses occurs before the
                     declaration of the input field length_of_name is
                     encountered. Even though the declaration of length_of_name
                     occurs on the same line as its use as a reference inside
                     the parentheses, the program is in error. */
        2 item_name char(length_of_name), 2 length_of_name dec(4),
        2 item_quantity dec(6);
```

DATA CONVERSION

If a numeric value occurs in a context where a character string is needed, the necessary conversion occurs automatically. The length of the converted string is the same as the quantity of nonblank characters that appear if the number is printed on a terminal. Similarly, if a character string occurs in a context where a number is needed, the conversion occurs automatically provided that the characters in the string are valid components of a number. (This topic is covered in depth in Section 6.)

These examples illustrate the rules governing data conversion. The use of ß in the second example stands for a space character.

1. Number occurs where character string needed:

Internal Number	Character String	String Length
123	123	3
4.56	4.56	4
-.007	-0.007	6

It is not possible to have a plus sign, or, leading or trailing zeros or blanks in the converted string.

2. Character string occurs where number needed.

Character String	String Length	Internal Number
987	3	987
654ßßß	6	654
ß32ß	4	32
ßß1.09ß	7	1.09
ßßßßßß-.006	11	-.006
0034.5600	9	34.56
+4	2	4
abc	3	error
d5	2	error
6e	2	error
7.8.9	5	error
+3+456	6	error
789-	4	error

The only valid characters in the character string to be converted are the decimal digits, a leading plus sign, a leading minus sign, and not more than one decimal point. Leading and trailing zeros and spaces are stripped off.

NUMBERING CONVENTIONS

In general, numbering of items begins with one, not zero:

Number of First Item	Item Description
1	Character positions within an input record.
1	Characer postions within a string.
1	Column positions within a report line.
1	Lines in a report page.
1	Sub-report numbers within a report.
1	Control break levels.
0	Phase numbers in the comments in the generated PL/I statements.

SECTION 4

NOTATION DEFINITION AND EXPLANATION

Several examples appear at the end of this section to assist the reader in understanding the notation used in later sections of this manual.

SYMBOLS USED IN GENERAL FORMATS

In addition to keywords and punctuation, the general format diagrams also contain brackets, braces, ellipses, and vertical double bars. Each of these items is discussed below.

Words and Phrases

Three types of English-appearing words and phrases exist in the general formats:

- WORDS IN ALL UPPERCASE LETTERS
- Words_With_Initial_Caps
- words_in_all_lower_case

Each of the uppercase and lowercase types identify a different kind of language element. The terms in ALL UPPERCASE LETTERS are usually one word. If the term involves more than one word, the words are separated by one or more white space characters. Most terms with initial caps and with all lowercase consist of more than one English word with the words connected by underline characters. A few of the initial cap terms and the all lowercase terms consist of one word.

ALL UPPERCASE

Text in all uppercase letters (e.g., DECLARE and PRINT) specifies keywords of the language. These words must be spelled exactly as shown in this text. Frequently, abbreviations exist for the English words. If the user chooses to select the abbreviation, then it must also be spelled exactly as shown.

There are no optional words within an MRPG phrase. Therefore, this manual does not distinguish keywords from optional words. Consequently, underlining of keywords is not used as it is in COBOL manuals.

INITIAL CAPS

Terms such as Declare_Input_File and Char_Expr are names of groups. These names can be thought of as abbreviations for groups.

The hyphen and digit at the end of an initial cap term (e.g., Char_Expr-3) is used to distinguish between different meanings of the information whose syntax is defined by another group. In COBOL manuals, the trailing identifiers in one general format have no relation to the trailing identifiers in another general format (i.e., in COBOL manuals, the numbering starts over at 1 again for each group). However, in this manual, the numbering carries across all groups. Lowercase terms with the same meaning have the same trailing identifier. Thus, the meaning for the value represented by Full_Expr-4 in the Footing group is the same as for the value represented by Full_Expr-4 in the Heading group.

ALL LOWERCASE

Terms such as "input_field_name" and "integer" identify places where the user must supply a name or a value. Each term specifies what type of information must be supplied. Thus, when "input_field_name" is encountered, the name written in the program at that point must be a name that is defined earlier in the program as the name of an input field. The term "integer" means that an integer must be supplied. Integers do not include decimal points. Therefore, supplying a number such as 7.3 is invalid. The applicable syntax and general rules clarify each situation.

The hyphen and digits at the end of the all lowercase term is the same as for initial cap terms (e.g., integer-19) and is used to distinguish between different meanings of the information that the user must supply. Thus, the meaning of "integer-13" in the Footing group is the same as "integer-13" in the Heading group.

Brackets and Braces

Brackets and braces have the same meanings as in COBOL documents:

- Brackets [] enclosing a portion of a general format indicate that either all of the options within the brackets may be omitted, or one but only one of the options within the brackets must be selected.
- Braces {} enclosing a portion of a general format indicate that one, but only one, of the options within the braces must be selected.

In both cases, options are stacked vertically within the braces or brackets. Occasionally, text is too long to fit onto one line -- in cases of this nature continuation is indicated by indenting the continuing lines a few column positions.

Braces and brackets always occur in balanced pairs. Further, the two matching items are the same height. When nesting of braces or brackets occurs, a brace or bracket is always higher than the next inward brace or bracket.

Ellipses

As in the general formats of COBOL, an ellipsis (...) represents the position at which repetition may occur at the user's option. The portion of the format that can be repeated is:

1. Select an ellipsis.
2. The brace or bracket that immediately precedes the ellipsis is the closing brace or bracket for the portion of the general format that can be repeated. Starting at this closing brace or bracket, scan to the left to locate the logically matching opening brace or bracket.
3. The ellipsis applies to that portion of the general format between these opening and closing braces or brackets.

Double Bars

MRPG permits some options to be used in any order. Further, some options may be used many times, but such multiple usage may be interlaced with the multiple usage of other options (several examples are presented later in this section). The MRPG notation is an extension of the notations:

- {‖options‖} as defined in Section 1.3.1.6 (May 76/76010) of the CODASYL Programming Language Committee COBOL Journal of Development. The brace and bar notation means that a selection of one or more of the options must be made, but the same sequence of words (option) must not be chosen more than once in that entry or statement.
- ‖options‖ as defined in Section 3.0.1 of the May 1977 draft of the CODASYL Data Description Language Committee DDL (Data Description Language) Journal of Development. The double bar notation means that at least one option must occur and at most, one of each option may occur.

An explanation of the MRPG notation is included after the figure. The letters X, Y, and Z are present to facilitate this discussion.

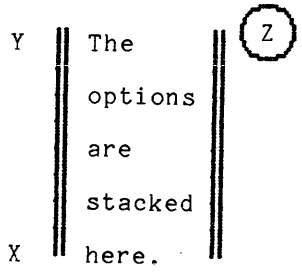


Figure 4-1. Generalized MRPG Double Bar Notation

The double bars mean that more than one of the options can be chosen. Further, when more than one option is chosen, they can be chosen in any order.

The letter X indicates the physical placement for the character that specifies the minimum quantity of options that must be selected from the set of options within the double bars. The value of X is either the digit zero or the digit one. If zero, all of the options can be omitted. If one, at least one of the options must be chosen.

The letter Y indicates the physical placement for the character that specifies the maximum quantity of times that each option can be selected. The value of Y is either the digit one or the letter "n". If one, any option that is selected can be selected only once. If "n", each option can be selected as many times as desired.

It should be noted that the X, which is associated with a minimum value, is placed physically lower in the figure than the letter Y, which is associated with a maximum value. Thus, the maximum-limit character is physically higher than the minimum-limit character.

The letter Z indicates the physical placement for the character that specifies the delimiter which is required between multiple options. The circle that the Z is within is part of the double bar notation. Usually, a space character is the between-options delimiter (i.e., the circle appears to be empty). Sometimes a comma is in the circle (i.e., a comma is required between options that are selected from the set of options within the double bars). The delimiting character is not a trailing character, but is a "between" character. (That is, the delimiting character is not used in front of the first option selected, nor is it used after the last option selected.)

Occasionally, the delimiting character is longer than one character. In such situations, the circle becomes two semicircles with the delimiter in-between the semicircles. See the Char_Expr general format diagram in Section 5.

EXAMPLES OF FORMAT NOTATION

These examples use words and options that are constructed for illustrative purposes only and bear no relation to the actual MRPG language.

1. Consider:

The set of all possible choices is:

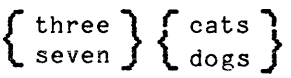
a. cat
b. dog
c. pig

2. Consider:

The set of all possible choices is:

a. omit everything
b. rat
c. hen
d. fox

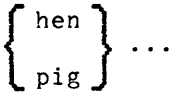
3. Consider:



The set of all possible choices is:

a. three cats
b. three dogs
c. seven cats
d. seven dogs

4. Consider:



There are an infinite number of possibilities. Several of the possible choices are:

a. hen
b. pig
c. hen pig
d. pig hen
e. hen hen hen hen pig hen hen pig

5. Consider:

The set of all possible choices is:

a. cat
b. dog
c. fox
d. cat dog
e. cat fox
f. cat dog fox
g. cat fox dog
h. dog cat
i. dog fox
j. dog cat fox
k. dog fox cat
l. fox cat
m. fox dog
n. fox cat dog
o. fox dog cat

6. Consider:

The only difference from the previous example is that the lower left digit is a zero, rather than a one. The set of all possible choices is the same as for the previous example except that there is one more choice, which is to omit everything.

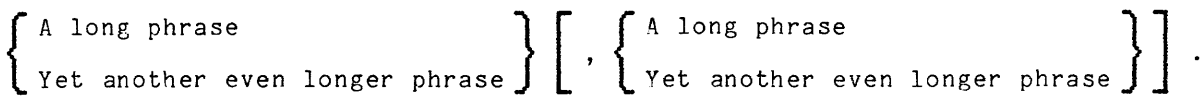
7. Consider:

There are an infinite number of possibilities, because there is an "n" at the upper left. Several choices are:

a. omit everything
b. hen
c. pig
d. hen,pig
e. pig,hen
f. hen,hen,pig,pig,pig,pig,hen,hen,hen,hen,hen,pig

Observe that no spaces are present either before or after the comma in the circle at the upper right of the double bars for this example. However, it is valid to have white space before, after, or both before and after the comma, because of MRPG's treatment of white space (i.e., white space between language elements is ignored).

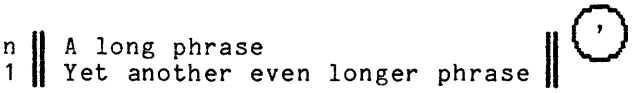
8. Consider:



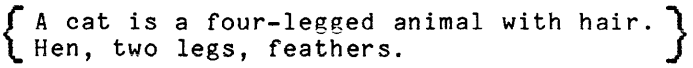
One of the possible choices is:

A long phrase, Yet another even longer phrase, Yet another even longer phrase, A long phrase, A long phrase

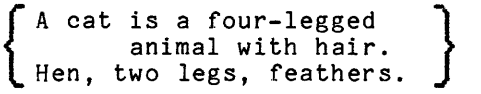
Inclusion of the above in a general format diagram consumes substantial horizontal space. The following double bar notation specifies the same rule in much less space.

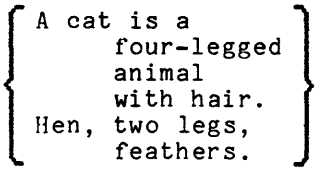


9. Consider:



Sometimes lengthy formats such as this, in combination with other portions of a general format do not fit on one line, hence text must be folded with indentation onto one or more continuation lines. The next two diagrams are equivalent to the above diagram.





SECTION 5

DEFINITION OF THE LANGUAGE

INFORMATION IN THIS SECTION

This section contains a complete, detailed definition of the language and includes the rules governing the writing of source programs. The 18 groups of information that specify the MRPG language are physically arranged in alphabetical order. Users with moderate to complete familiarity with the MRPG language can write most programs with the aid of Appendix D only. That appendix contains all of the general format diagrams arranged in hierarchal order. When a detailed explanation of a particular area is required, the alphabetical arrangement of the groups in Section 5 facilitates speedy location of the desired information.

This section is physically organized as: 1) a skeleton of the material within each group, 2) a set of rules that apply to all groups, and 3) the 18 groups discussed in alphabetical order.

MANDATORY GROUP SEQUENCE

Several of the groups must appear in a specific sequence whenever these groups are present in a program. Some of the groups may be repeated, but the sequence shown below must be preserved. Indentation represents a subservient group.

```
The_MRPG_Program
     Declare_Parameters
     Declare_Input_File
          Input_Field_Def
     Declare_Variable
     Define_Report
          Report_Control
          Heading
               Line
                    Report_Field_Def
          Detail
               Line
                    Report_Field_Def
          Footing
               Line
                    Report_Field_Def .
     Execute_Phase
```

Figure 5-1. Mandatory Group Sequence In A Program

A semiboxed heading appears at the top of each page, giving the name of the group being discussed on that page. The first time that this manual is read, the reader may find it desirable to read the 18 group discussions in the order:

The_MRPG_Program
: the overall structure of a program

Declare_Parameters
: declare all parameters to be used including their attributes

Declare_Input_File
: identify the input file and its fields

Input_Field_Def
: specify the attributes of one input field

Declare_Variable
: specify the name and attributes of one variable

Define_Report
: identify a report

Report_Control
: specify several major properties of a report

Heading
: specify controls for report, page, and detail heading lines

Detail
: specify controls for a set of detail lines

Line
: specify line controls for one line including fields in the line

Report_Field_Def
: specify the value and format for one field

Footing
: specify controls for detail, page, and report footing lines

Execute_Phase
: specify the order in which sorts, lines, and reports are done

Full_Expr
Relationship_Test
Char_Expr
Char_Ref
Arith_Expr

} used to form expressions and make tests

```
The_Name_Of_The_Group                    The_Name_Of_The_Group

Group Name: The_Name_Of_The_Group

     A few lines of text here summarize the general purpose of this
portion of the language.

General Format:

     A diagram that specifies a portion of the MRPG language in a
COBOL-like notation. Most diagrams reference other groups.

Examples:

     One or more examples illustrating specific points of the general
format.

Syntax Rules:

     Several numbered paragraphs that define, clarify, or restrict the
exact manner in which this portion of a program must be written.

General Rules:

     Several numbered paragraphs that define, clarify, or restrict the
meaning, content, and structure of the variable items shown in the
general format diagram.
```

Figure 5-2. Skeleton for Section 5

RULES APPLICABLE TO ALL GROUPS

The remainder of this section consists of the general formats and includes related examples, syntax rules, and general rules. The following paragraphs apply to every group. They are stated here so that they need not be repeated in the discussion of each group.

1. A term formed from words with initial capitals connected by underline characters (e.g., Declare_Parameters) refers to a group. Think of a group name as being an abbreviation for that group. The overall effect is to perform a group explosion, analogous to a parts explosion.

2. The dash and number (e.g., -5) at the end of a term serve to identify terms with unique meanings. Terms ending in -0 identify the point at which the all_lower_case term is "defined" (i.e., the point at which the properties and characteristics associated with the all_lower_case term are established). Terms ending with other than -0 are places in which the term is used.

3. Within the examples in this section, the following names and character strings have these meanings:

   your_mrpg_os
   : The name of the MRPG object segment. Thus, this is the character string typed to invoke the MRPG-OS from Multics command level. The MRPG-OS is assumed to be in the current working directory.

   your_input
   : The name of the segment that contains the input file. It also is assumed to be in the current working directory.

   report
   : The name of a report.

   your_output
   : When the report is written into a segment, this is the name of that segment. It too is assumed to be in the current working directory.

4. Within the examples, unless stated otherwise, assume that the MRPG-OS is invoked by the user typing on a terminal at command level and that messages sent to user_output and to error_output are also typed on the terminal.

5. Unless stated otherwise, the maximum length of a character string is 256 characters.

6. Unless stated otherwise, the characters used in a character string may be chosen from the full (128) ASCII character set. However, the NUL character, octal 000, and the PAD character, octal 177, cannot appear in a canonical string. (Refer to Section 3 and Appendix A of the MPM Reference Guide.)

7. An IF test examines the value of an expression and either succeeds or fails. The type of data in the result of the expression being tested can be numeric, character, or boolean.

   If the expression result type is numeric, the IF test fails when the result is zero; otherwise, the IF test succeeds.

   If the expression result type is character, the IF test fails when the character string is the five characters FALSE; otherwise the IF test succeeds. The FALSE string may be spelled with any mixture of uppercase and lowercase letters (e.g., FALSE, false, FaLsE, fALSe, etc.)

   If the expression result type is boolean, the IF test succeeds when the boolean value is true and fails when the value is false.

8. When referring in text to the keywords that identify an option, and there are both a long form and a short form of the keyword, only the long form is used in the text. It is understood that the remarks apply to the short form also. Thus,

   The CHARACTER option ...

   is used instead of

   The CHARACTER or CHAR option ...

9. When data is stored in a character string variable, the left-hand end of the data string is positioned at the left-hand end of the variable's storage area. If the data is too long to fit, the right-hand portion of the data is truncated and discarded. If the data is shorter than the variable's storage area, and the variable was declared with the VARYING keyword, the length of the new value is set to the actual length of the data. However, if the declaration does not include the VARYING keyword, the data is padded on the right with spaces. The length remains whatever length was declared for the variable. For example, assign the data 12345678 in each case below. The overstruck character Ø represents a space.

Variable Type	Declared Length	Result Value	Result Length
Non-varying	6	123456	6
Non-varying	10	12345678ØØ	10
Varying	6	123456	6
Varying	10	12345678	8

10. All names made up by the user must be unique. That is, the user chooses character strings for these types of identifiers:

```
parameter_name
input_field_name
local_variable_name
set_variable_name
table_variable_name
report_name
detail_name
```

Think of the identifiers for all of these types of names as being in one set. All members of that set must be unique.

Arith_Expr

Group Name: Arith_Expr

The Arith_Expr group and its Arith_Ref subgroup provide arithmetic operators and parentheses so that conventional arithmetic expressions may be formed. In addition to the details discussed here, Section 7 contains a unified treatment of the interactions between the rules stated in the Full_Expr group and its subsidiary groups.

General Format:

```
[ + ]                [ { + }                ]
[ - ] Arith_Ref      [ { - } [ + ]          ]
                     [ { * } [ - ] Arith_Ref ] ...
                     [ { / }                ]
```

where Arith_Ref is

```
{ number-4                                                    }
{ input_field-name-6                                          }
{ local_variable_name-4                                       }
{ parameter_name-1                                            }
{                                                             }
{ %PAGENUMBER ( [ report_name-1 ] )                           }
{                                                             }
{ TRANSFORM ( Full_Expr-12, table_variable_name-1 )           }
{                                                             }
{ ( Full_Expr-13 )                                            }
```

Examples:

1. Simple arithmetic expressions.

```
  5
+ 5.7
- 6.3
  name_of_an_input_field + name_of_a_local_variable
  shop_cost * ( 1 + overhead_factor )
  unit_price * ( 1 + 3 * (burden_1 + burden_2)/factor_3)
```

2. Provide the current page number in the page heading line of a report.

```
define 1 report parts_analysis ...
        2 pagehead,
         3 line,
          4 "PARTS ANALYSIS" col 11,
          4 %mmddyy col 31,
          4 "Page" col 51,
          4 %pagenumber (parts_analysis) col 56;
```

Arith_Expr

3. Use of a transform variable.

```
declare rank_word char(20) varying;
declare rank_code_to_rank_name table
      ( 1 -> "letter" 2 -> "word" 3 -> "sentence");
   .
   .
   .
rank_word := transform ( r_count, rank_code_to_rank_name);
```

If r_count = 2, rank_word is set to "word" (without the quote characters).

4. Examples using prefix (unary) arithmetic + and - operators.

   a. - count_3

   b. alpha ++ beta -- gamma *- delta +- epsilon

      is equivalent to:

      alpha + beta - (gamma * delta) - epsilon

5. The following numerical examples illustrate the meanings of the infix (binary) arithmetic operators.

Expression	Result
1.2	+ 1.2
+3.4	+ 3.4
-5.6	- 5.6
7 + 8	+ 15
7 + 8 -3	+ 12
(7 + 8 - 3) * -3	- 36
-(7 + 8 - 3) * -3 / -10	- 3.6

Syntax Rules:

1. Parentheses may be nested to any depth.

2. If only one report is defined with the DEFINE 1 REPORT keywords from the Define_Report group, report_number-1 may be omitted from the %PAGENUMBER option.

General Rules:

1. The %PAGENUMBER builtin function returns the integer, in a character string varying form, of the current page number of the report specified by report_name-1. If report_name-1 is omitted, the current page number of the one and only report that was defined is returned.

2. When the TRANSFORM option is specified, Full_Expr-12 is evaluated. The resulting value is searched for in the first members of the pairs of values that were declared for the table variable specified by table_variable_name-1. The value of Full_Expr-12 need not have an integral value.

Arith_Expr

3. The + and - operators that immediately precede the Arith_Ref term in the general format diagram are the conventional prefix (or unary) arithmetic operators with these meanings:

Operator	Meaning
+	Plus; use the value of Arith_Ref as is.
-	Minus; use the negative of the value of Arith_Ref; multiply the value of Arith_Ref by minus one.
Neither	Same as for +

4. The set of + - * / operators in the middle of the general format diagram are the conventional infix (or binary) arithmetic operators, with these meanings:

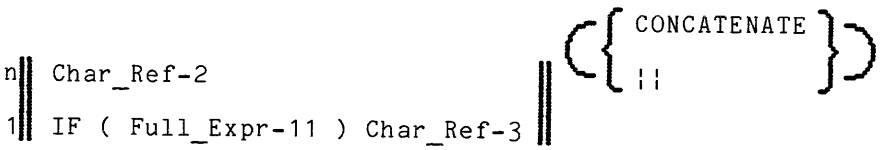
Operator	Meaning
+	Add
-	Subtract
*	Multiply
/	Divide, with any remainder included in the result.

Char_Expr

Group Name: Char_Expr

The Char_Expr group forms a character string from shorter character strings. (Section 7 contains a unified treatment of the interactions between the rules stated in the Full_Expr group and its subsidiary groups.)

General Format:



Examples:

Assume that:

- Today is Wednesday, 1980 December 31
- The parameter color contains blue as its value
- The local variable shape contains triangle as its value
- The input field part_number contains PHX23B7 as its value
- The local variable alpha contains FIRST as its value

1. The expression:

   "This is the " ||%day|| " report."

   yields:

   This is the Wednesday report.

2. The expression:

   "part." ||color|| "." ||shape

   yields:

   part.blue.triangle

   which might be useful as a file name for a report.

Char_Expr

3. The expression:

```
%substr (part_number ||color, 4, 7)
```

yields:

```
23B7blu
```

4. The expression:

```
if (color = "blue") alpha
```

yields:

```
FIRST
```

5. The expression:

```
"elephant_" ||
if ( shape = "square" ) "cat" ||
if ( %day = "Saturday" ) "hen" || "-COW-" ||
if ( "red" = color ) "duck" || "_HORSE"
```

yields:

```
elephant_-COW-_HORSE
```

6. This example shows several ways of arriving at the name or value of a field. Assume that:

   - The current time is 2.3 seconds before the end of 1980
   - The value of the parameter bogie is 43
   - The value of the part_cost field in the current input record is 410

   The line:

   ```
   "Today is " || %mmddyy || " at " || %hhmmss
   ```

   yields:

   ```
   Today is 12/31/80 at 23:59:57
   ```

   The line:

   ```
   "Today is " || %month || " " || %substr( %mmddyy, 4,2)
   ```

   yields:

   ```
   Today is December 31
   ```

   The line:

   ```
   part_cost * 1.1 + bogie
   ```

   yields:

   ```
   494      /* 410 * 1.1 + 43 results in 494 */
   ```

Syntax Rules:

1. The semicircles, CONCATENATE, and || construct at the upper right of the general format diagram means that options within the double bars may be strung together with the string CONCATENATE or || between the options.
2. At least one Char_Ref-2 must be supplied. An indefinite number of additional Char_Ref-2 items may be supplied.
3. Every IF test must have an accompanying Char_Ref-3.
4. When IF (Full_Expr-11) phrases appear, an indefinite number of Char_Ref-2 and/or Char_Ref-3 values may be concatenated together to yield the final character string for the Char_Expr.

Char_Expr

General Rules:

1. The values of Char_Ref-2 and/or Char_Ref-3 are treated as character strings.

2. If a Char_Ref is an arithmetic value, it is converted into a character string and then the concatenation occurs.

3. If a Char_Ref is a boolean value, it is converted into a character string and then the concatenation occurs. A boolean true value is converted to the 4-character string "true" while a false value is converted to the 5-character string "false".

4. When an IF clause appears, the value of Full_Expr-11 is tested. The character string result for this portion of the Char_Expr is the value of Char_Ref-3 should the test succeed and is null otherwise.

Group Name: Char_Ref

The Char_Ref group provides several ways to construct character strings and provides builtin functions for obtaining information and manipulating character strings. In addition to the details discussed here, Section 7 contains a unified treatment of the interactions between the rules stated in the Full_Expr group and its subsidiary groups.

General Format:

```
{ Arith_Expr-1                                                    }
{ "string-9"                                                      }
{ %MMDDYY                                                         }
{ %YYDDD                                                          }
{ %MONTH                                                          }
{ %DAY                                                            }
{ %HHMMSS                                                         }
{ %SUBSTR ( Char_Expr-9, Arith_Expr-2 [ , Arith_Expr-3 ] )        }
{ %REPEAT ( Char_Expr-10, Arith_Expr-5 )                          }
```

Examples:

Assume that:

- Today is Wednesday, 1980 December 31
- The time is 2.3 seconds before midnight (i.e., 57.7 seconds of the minute has passed)
- The local variable j_count has 5 as its value

1. Arithmetic expressions.

```
27                yields 27
j_count           yields  5
j_count *3 + 6.9  yields 21.9
```

Char_Ref

2. Quoted strings. The quotes are not part of the value. The length of the value is equal to the quantity of characters between (i.e., exclusive of) the quote characters. A quote can be included in a string by doubling the quote. A Ƀ denotes a space character.

Appearance in source program	Length	Value
"apple"	5	apple
"yellowɃbanana"	13	yellowɃbanana
"spaceɃatɃendɃ"	13	spaceɃatɃendɃ
"""orange"""	8	"orange"
"""catɃ""Ƀdog"""	11	"catɃ"Ƀdog"
"cowɃ""""Ƀpig"	10	cowɃ""Ƀpig
""	0	

The last line of the above table is a null string.

3. Builtin functions provide date and time information. The example assumes that the time in history is New Year's Eve - 1980.

```
%mmddyy   yields   12/31/80
%yyddd    yields   80366
%month    yields   December
%day      yields   Wednesday
%hhmmss   yields   23:59:57
```

4. Portions of a string may be obtained with the substring builtin function.

```
%substr ("abcdef", 1, 2)                    yields ab
%substr ("abcdef", 4)                       yields def
%substr (%month, 1, 3)                      yields Dec
%substr (%substr (%hhmmss, 3, 4),2,2)       yields 59
%substr ("1234567890", j_count,j_count)     yields 56789
```

5. Repeat character string.

```
%repeat ("ab", 3)                yields  ababab
%repeat (j_count, j_count)       yields  55555
%repeat (%substr(%yyddd,4,2),3)  yields  656565
```

Syntax Rules:

1. Arith_Expr and Char_Expr items may be complex statements. There is no specific restriction on the degree of complexity or nesting of parentheses.

2. Builtin functions may be nested to any depth.

3. The maximum length of the resulting string is 256 characters, unless the intended use imposes a smaller maximum.

Char_Ref

General Rules:

1. If the "string-9" option is chosen, the terminating quote character is the first unpaired quote character occurring after the initial quote character (see Example 1).

2. The date and time functions return values that are obtained during the execution of MRPG-OS. The system calendar clock is interrogated (once) shortly after MRPG-OS begins execution. Printing the time on more than one occasion during the execution of MRPG-OS yields the same value.

3. The %MMDDYY built-in function returns the date as an eight character string.

Length	Example	Description
2	12	Number of the month in the year.
1	/	A right slant (or slas) character.
2	31	Number of the day in the month.
1	/	A right slant.
2	80	Number of the year in the century, starting with 00.

4. The %YYDDD built-in function returns the date as a five character string.

Length	Example	Description
2	80	Number of the year in the century, starting with 00.
3	366	Number of the day in the year, starting with 1.

5. The %MONTH built-in function returns the unabbreviated name of the current month as a varying character string with the initial letter in uppercase and the remaining letters in lowercase.

6. The %DAY built-in function returns the unabbreviated name of the day of the week as a varying character string with the initial letter in uppercase and the remaining letters in lowercase.

Char_Ref

7. The %HHMMSS built-in function returns the time of day as an eight character string.

Length	Example	Description
2	23	Number of the hour of the day, starting with 00 after midnight.
1	:	A colon character.
2	59	Number of the minute of the hour
1	:	A colon character.
2	57	Number of the second of the minute.The actual time in seconds is truncated to yield an integer value.

8. The %SUBSTR built-in function is identical to the PL/I substring built-in function. Char_Expr-9 is the character string examined. Arith_Expr-2 specifies the number of the first character of Char_Expr-9 as the first character of the result. Arith_Expr-3 specifies the quantity of characters in Char_Expr-9 that constitutes the substring. If Arith_Expr-3 value is zero, the result is a null string. If Arith_Expr-2 or Arith_Expr-3 is not an integer, the value is truncated to an integer. An error occurs if Arith_Expr-2 or Arith_Expr-3 is negative, or if the sum of the truncated values of Arith_Expr-2 and Arith_Expr-3 is larger than the length of the Char_Expr-9 string. If Arith_Expr-3 is omitted, the substring ends at the end of the Char_Expr-9 string. This is a PL/I characteristic.

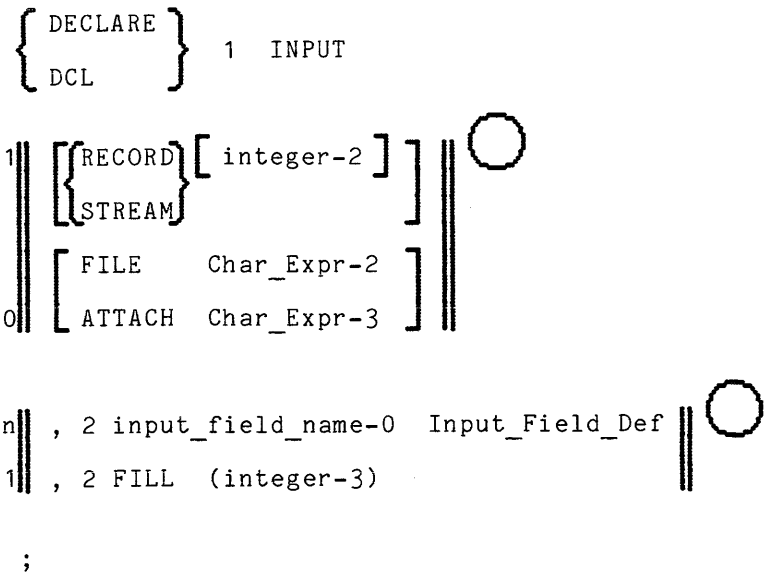
9. The %REPEAT built-in function returns a character string in which the value of Char_Expr-10 is repeated the number of times that is equal to the value of Arith_Expr-5. If the value of Arith_Expr-5 is not an integer, the value is truncated to an integer. An error condition results if the truncated value is negative. If the truncated value is zero, the result is a null string.

Declare_Input_File

Group Name: Declare_Input_File

The Declare_Input_File group, along with its subsidiary groups, provides information concerning the type, location, and structure of the input file.

General Format:



Examples:

These examples concentrate on the overall file characteristics. See "Input_Field_Def" group for examples of individual fields.

1. The input file is a segment containing several records. Each record ends with a newline character.

```
dcl 1 input stream file "your_input",
```

2. Obtain the same file as in Example 1 (using the ATTACH phrase) to illustrate the ATTACH phrase.

```
dcl 1 parameter,
    2 file_name char(*);
dcl 1 input stream attach "vfile_ "||file_name,
```

Declare_Input_File

Syntax Rules:

1. If RECORD or STREAM is not specified, the default is STREAM.

2. If the FILE or ATTACH option is not specified, the MRPG-OS cannot run as a command. It can however run as an I/O appendage via the report_ I/O module. Usually, this is done in conjunction with LINUS.

3. If MRPG-OS is run as an I/O appendage, all FILE and ATTACH phrases are ignored.

General Rules:

1. If STREAM is chosen, each record in the input file is assumed to be followed by a newline character. The newline character is not part of the record. If integer-2 is omitted, the maximum record length is assumed to be 500.

2. RECORD indicates that every record is the same length. IF RECORD is chosen, but integer-2 is omitted, the length of each input record is assumed to be equal to the sum of the lengths of the individual fields. Thus, the last character of record N is immediately followed by the first character of record N+1.

3. If RECORD is chosen and integer-2 is supplied, the length of each record is assumed to be equal to the value of integer-2. This allows the user to omit the declarations of fields not used and which are located at the end of each record.

4. With STREAM, the opening mode for the input file is stream_input. With RECORD, the opening mode is sequential_input.

5. Char_Expr-2 must be a character string. It is used by the vfile_ I/O module as the relative or absolute pathname of a segment for the file. The MRPG language does not impose any constraints on the characters in the string. However, the intended use of this string does impose constraints. Allowable characters and the length of the string are restricted to what is allowed in relative pathnames of segments. (See "Section 3" of the MPM Reference Guide.)

6. Char_Expr-3 is a character string used as an attach description for an I/O module, usually the vfile_ module. The ATTACH keyword must be supplied to use an I/O module other than the vfile_ I/O module. (Refer to the MPM Subroutines, MPM Subsystem Writers' Guide, or the MPM I/O manuals for details of the required attach description.)

7. The FILL declaration is used to skip over data characters in the input record. The value of integer-3 specifies how many data characters to skip. Any field length and/or field delimiter information in the record is automatically skipped.

Declare_Parameters

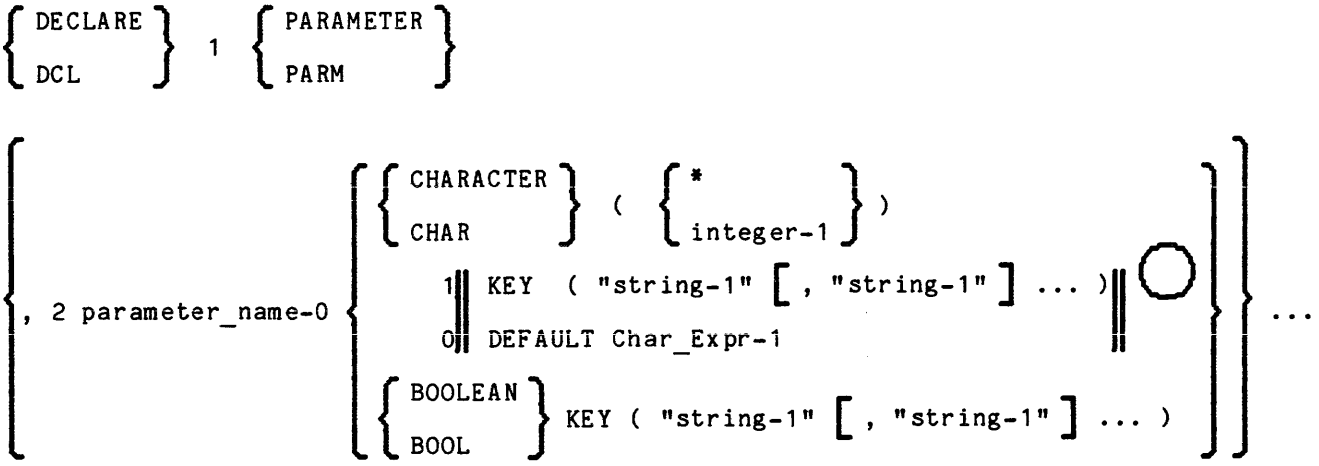
Group Name: Declare_Parameters

The Declare_Parameters group provides the capability to:

- Specify that parameters may be supplied
- Describe their acceptable forms

General Format:

In the general format diagram, the term KEY is a keyword. Following KEY, string-1 specifies a key_string that is to be supplied when the MRPG-OS is invoked. At that time, a key_value is supplied immediately after the key_string in the command line. The key_value then becomes the value of the parameter.



Examples:

1. Declare and supply one parameter, without using a KEY phrase.

   ```
   dcl 1 parm, 2 city char(*)
   ```

   Typing the command line:

   ```
   your_mrpg_os Phoenix
   ```

   assigns the value Phoenix to the parameter named city.

Declare_Parameters

2. Declare and supply two parameters that must be supplied in a specific sequence.

```
dcl 1 parm,
    2 city char(*),
    2 state char(*);
```

Typing:

```
your_mrpg_os Phoenix Arizona
```

assigns the value Phoenix to city and Arizona to state. However, typing:

```
your_mrpg_os Texas Austin
```

assigns Texas to city and Austin to state.

3. Declare and supply two parameters using key phrases (provides input sequence independence).

```
dcl 1 parm,
    2 city char(*) key("-city"),
    2 state char(*) key("-state");
```

Typing either:

```
your_mrpg_os -city Chicago -state Illinois
```

-OR-

```
your_mrpg_os -state Illinois -city Chicago
```

yields the same result.

4. Using default values.

```
dcl 1 parm,
    2 input_file char(*) key ("-input") default "your_input";
```

Typing either:

```
your_mrpg_os -input your_input
```

-OR-

```
your_mrpg_os
```

gives the same result. However, typing:

```
your_mrpg_os -input my_input
```

assigns the value my_input to the input_file parameter.

Declare_Parameters

5. Control the length of a parameter's value.

```
dcl 1 parm, 2 color char(4);
```

Typing:

```
your_mrpg_os blue     assigns the color value "blue"
your_mrpg_os yellow   assigns the color value "yell"
your_mrpg_os red      assigns the color value "red "
```

6. Make flexible the sequence in which arguments are typed.

```
dcl 1 parm,
    2 animal char(*) key ("-a", "-animal"),
    2 vegetable char(*),
    2 mineral char(*),
    2 gas char(*) key ("-g"),
    2 liquid boolean key ("-water");
```

All of the following lines give the same result.

```
your_mrpg_os -a cat carrot granite -g helium -water
your_mrpg_os carrot granite -g helium -water -a cat
your_mrpg_os -g helium carrot -animal cat -water granite
```

Since neither the vegetable nor the mineral parameter declarations contain a key phrase, the first nonkeyed argument is associated with vegetable and the second nonkeyed argument is associated with mineral.

Syntax Rules:

1. The KEY/DEFAULT phrases are a continuation of the CHARACTER clause.

2. The string-1 values for all of the parameters form a set of quoted strings. There must be no duplicates in that set.

3. Parameter key_strings and key_values cannot contain semicolons, parentheses, or brackets, unless such characters are contained within quoted strings. (These characters have special meaning to the command processor.) The key_strings and key_values are encountered by the command processor when the MRPG-OS is invoked.

General Rules:

1. CHARACTER (*) means the length of the value associated with this parameter is the length of the argument sent to the MRPG-OS.

2. CHARACTER (integer-1) means the value assigned to the parameter contains the quantity of characters specified by the value of integer-1. If more than integer-1 characters are supplied as an argument value, only the first integer-1 characters are used as the parameter value. If less than integer-1 characters are supplied, sufficient spaces are appended to the supplied characters to yield a string containing integer-1 characters.

Declare_Parameters

3. If the KEY phrase is included in the declaration of some CHARACTER parameters, then the key_strings and their key_values may be supplied in any sequence. Those CHARACTER parameters whose declarations do not include a KEY phrase are assigned values from the set of nonkeyed arguments. The first nonkeyed argument is assigned to the first parameter in the set of parameter declarations whose declaration does not include a key phrase, the next nonkeyed argument to the next nonkeyed parameter, etc.

4. The first character of string-1 must be a dash (minus sign, hyphen, or octal 055).

5. Multiple forms of a key_string are specified by including multiple string-1 items in a key phrase.

6.! The end result of evaluating Char_Expr-1 must be a quoted string. If no value is supplied for a parameter whose declaration includes a DEFAULT phrase, the value of Char_Expr-1 is assigned to that parameter.

7. A BOOLEAN parameter has the value TRUE if any of the string-1 keys specified for that parameter are present in the arguments sent to the MRPG-OS. If none of the string-1 keys are present in the argument list, then the parameter has the value FALSE.

8. When MRPG-OS is invoked with arguments, each argument is examined to determine whether or not it begins with a dash. If not (no dash), then it is treated as a nonkeyed argument (see "General Rule 3"). However, if the first character of the argument is a dash, the argument is treated as a key_string. The MRPG-OS attempts to locate the key_string in the set of string-1 values specified in the KEY phrases.

   Note: A negative decimal value as an argument is invalid.

9. The value of a parameter cannot be changed by MRPG-OS.

10. When the MRPG-OS is invoked, all CHARACTER parameters which do not have a DEFAULT must be supplied.

Declare_Variable

Group Name: Declare_Variable

The Declare_Variable group is used to establish variables (not in the input file) for use in calculations, tests, and reports.

General Format:

```
{ DECLARE }
{ DCL     }

{ local_variable_name-0 { DECIMAL                                                  }
{                       { DEC                                                      }
{                       { { CHARACTER } ( integer-7) [ VARYING ]                   }
{                       { { CHAR      }              [ VAR     ]                   }
{                       { BOOLEAN                                                  }
{                       { BOOL                                                     }
{
{ set_variable_name-0 SET ( { number-1 [ , number-1 ] ... } )
{                           { "string-5"[ , "string-5"] ... }
{
{ table_variable_name-0 TABLE ( { number-2    -> number-3    } ...) [ VARYING ]
{                               { number-2    -> "string-7"  }       [ VAR     ]
{                               { "string-6"  -> number-3    }
{                               { "string-6"  -> "string-7"  }
;
```

Examples:

1. Variables used in conventional arithmetic calculations are:

```
dcl   count dec;
dcl   grand_total dec;
dcl   average dec;
 .
 .
 .
average := grand_total / count;
```

Declare_Variable

2. Character string variables are:

```
dcl date_yymmdd char (6);
/* e.g., 770929 */
dcl date_year_month_day char (17) varying;
/* e.g., 1977 September 29 */
 .
 .
 .
date_yymmdd :=  %substr (%mmddyy,7,2)
              ||%substr (%mmddyy,1,2)
              ||%substr (%mmddyy,4,2);
date_year_month_day :=  "19"||%substr (%yyddd,1,2)
                      ||" "||%month||" "
                      ||%substr(%mmddyy,4,2)
```

3. Boolean variables used in a test could be:

```
dcl test_3 boolean;
 .
 .
... let (test_3 := true;)...
 .
 .
 .
if (test_3) then ...
```

4. A data set can be established for later use in determining whether or not a value is a member of the set.

```
dcl all_the_dec_digits set (0,1,2,3,4,5,6,7,8,9);
dcl state_name_western set
       ("California", "Oregon", "Washington", "Hawaii",
        "Alaska");
 .
 .
 .
if deduction_code not in all_the_dec_digits
     then error_message := "Invalid deduction code.";
fi;
if state_name in state_name_western
     then sales_office := "San Francisco";
fi;
```

Declare_Variable

5. Values may be encoded or decoded with the aid of a table variable.

```
dcl part_no_to_price table
     (1111 -> 27.50 2222 -> 49.98 3333 -> 67.23);

dcl rank_code_to_rank_name table
     (1 -> "letter" 2 -> "word" 3 -> "sentence" 4 ->
     "paragraph");

dcl state_code_to_number table
     (AL" -> 1 "AK" -> 2 "AZ" -> 3 ... "WY" -> 50);

dcl state_name_to_code table
     ("Alabama" -> "AL" ... "Wyoming" -> "WY");
  .
  .
  .
unit_price := transform (part_number, part_no_to_price);

state_number := transform
     (transform (state_name, state_name_to_code),
      state_code_to_number);
```

Syntax Rules:

1. The Declare_Variable group is used once for each variable that is neither a parameter nor a field in the input file.
2. Each occurrence of the Declare_Variable group ends with a semicolon.
3. When a TABLE variable is defined, one of the four possible options is selected. Only that option form may be used for that variable (i.e., the option forms may not be mixed in the declaration of a TABLE variable).

General Rules:

1. Local variables are not automatically initialized. One-time initialization is covered in the discusssion of the BEGIN keyword in the Execute_Phase group. Once-per-control-break initialization can be done in the DETAILHEAD portion of the Heading group.
2. The DECIMAL option specifies a numeric variable.
3. The CHARACTER option specifies a character string variable.
4. If CHARACTER is selected but VARYING is omitted, integer-7 specifies the quantity of the characters that are always occupied by the character string.
5. If CHARACTER is selected along with VARYING, integer-7 specifies the maximum length of the string. The string's length may be less at times.
6. A variable declared with BOOLEAN can have either a true or a false value.

Declare_Variable

7. The SET declaration is used to establish a set of numbers or a set of character strings for use in later parts of the program. Tests may be made later to determine whether or not a data item is in a data set that was established with the SET option. A set of numbers may contain any mixture of integers and non-integers. All numbers are considered to be positive numbers. All ASCII characters are permitted in string-5. The values for a SET variable cannot be changed during the execution of the MRPG-OS.

8. The TABLE declaration provides a means of converting a value to some other value via a table lookup. When used, the supplied value is searched for in the first parts of each transform pair (i.e., examine the number-2 and the string-6 values). If the value is found, the second part of that pair is made available (i.e., the number-3 or the string-7 value). If the VARYING or VAR option is selected, the length associated with a string is the actual length of that string. If the VARYING or VAR option is omitted, all strings have the same length. The length is the length of the longest string supplied in the declaration of this table variable. A number may be either an integer or a non-integer. All numbers are considered to be positive numbers. All ASCII characters are permitted in string-6 and string-7. The values for a TABLE variable cannot be changed during the execution of the MRPG-OS.

Define_Report

Group Name: Define_Report

The Define_Report group defines the overall structure of a report. This group may occur several times to define multiple reports.

General Format:

```
DEFINE 1 REPORT report_name-0
[ Report_Control ]
[ Heading ]
{ Detail } ...
[ Footing ]
;
```

Example:

See Section 2 and Appendix B for complete examples that include the Define_Report group.

Syntax Rules:

1. Note that only the Detail group is required.
2. A semicolon ends the Define_Report group. Other semicolons may exist within the group.
3. Only the Detail group may occur more than once in the definition of a report.
4. Whichever groups are used, they must be used in the order shown above.

General Rules:

1. The Report_Control group is used to supply overall report control information when the default values and actions are other than what is required.
2. The Heading group is used if report, page, and/or detail heading lines are required.
3. The Detail group is used to supply the specific information about the content of each detail line.

Define_Report

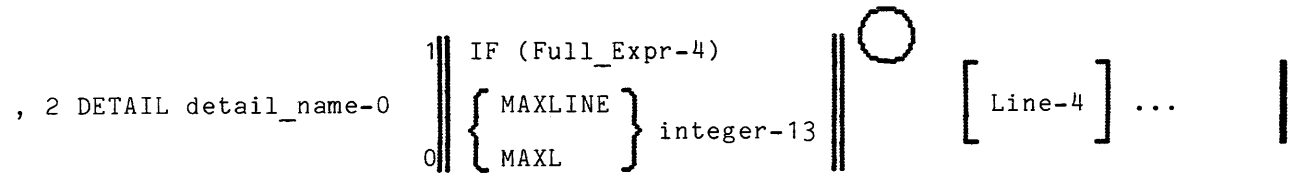
4. The Footing group is used if detail, page, and/or report footing lines are required.

Detail

Group Name: Detail

The Detail group associates a name with a detail line and specifies tests to determine whether or not the line is to be produced and whether or not it is produced for the current page or the next page.

General Format:



Examples:

1. Unconditionally print the line.

```
define 1 report name_address_list
                  on file "your_output",
       2 detail name ...
 .
 .
 .
print name;
```

2. Print the line only if some condition is satisfied.

```
2 detail name if (dept_code > 123) ...
 .
 .
 .
```

Syntax Rules:

1. More than one line can be defined as part of a DETAIL group. If this is done, the detail_name-0 refers to all of those lines. It is not necessary however, to define any lines.

2. Examination of the Define_Report group shows that the Detail group may be used several times in the definiton of a report (each with a different detail_name).

Detail

General Rules:

1. The DETAIL group specifies a set of detail lines. When an IF test is specified, Full_Expr-4 is evaluated and the result is tested. Assuming that the IF test succeeds, the set of lines becomes a candidate for being produced. In the event that the IF test fails, the set of lines is omitted. When no IF test is specified, the set of lines is a candidate for being produced. If the set of lines becomes a candidate for being produced, then a similar IF test within each line definition may be specified to determine whether or not each line is to be produced. (See "Lines" group for details.)

2. This paragraph and the next three general rules occur with almost identical wording in the Heading and Footing groups. Small changes are made because the group name changes.

3. A MAXLINE value, known as report_maxline, is established for the report as a whole in the Report_Control group. If, in this Detail group, the MAXLINE integer-13 phrase is supplied, the value of integer-13 must be less than or equal to the value of report_maxline.

4. The line number of the highest-numbered line on which the first line of the DETAIL set of lines may be printed is the minimum of report_maxline and integer-13, if the MAXLINE integer-13 phrase is supplied. If the MAXLINE integer-13 phrase is omitted, then report_maxline specifies the line number of the highest-numbered line on which the first line of this set is produced.

5. If more than one line is defined for the DETAIL set of lines, then the value of integer-13 must not be larger than report_maxline minus the maximum quantity of lines that might be produced for the DETAIL set being specified. As an example, suppose that the value of report_maxline is 45. Assume that the DETAIL set of lines has six lines specified, but an IF test is specified for each line. Further assume that the user knows that the IF tests and the data in the input file are such that no more than four of the six lines are ever produced on the same page. Then, a value of 41 can be supplied for integer-13.

Group Name: Execute_Phase

The Execute_Phase group is the executable portion of the program. This is the only group that executes sorting and printing.

General Format:

Format 1: (Valid only for the first phase.)

```
BEGIN ( [ local_variable_name-2 := Full_Expr-9 ; ] ... )

                        [       n|| INPUT                  ||  (,)      ]
[ Loop_Statement ] ...  [ HOLD   || input_field_name-4     ||        ;  ]
                        [       0|| local_variable_name-3  ||           ]
```

Format 2: (Valid for all phases after the first phase.)

```
BEGIN ( [ local_variable_name-2 := Full_Expr-9 ; ] ... )

[      n || { input_field_name-5    } [ ASCENDING  ] ||  (,)  [    { DUPLICATE } ]   ]
[ SORT   || {                       } [ ASC        ] ||       [ NO {           } ] ; ]
[      1 || { local_variable_name-4 } [ DESCENDING ] ||       [    { DUPL      } ]   ]
[                                     [ DESC       ]                                 ]

{ Loop_Statement } ... [ HOLD ; ]
```

where Loop_Statement is:

```
{ { input_field_name-4    }                                                 }
{ {                       } := Full_Expr-8 ;                                }
{ { local_variable_name-2 }                                                 }
{                                                                           }
{ PRINT { report_name-1 } ;                                                 }
{       { detail_name-1 }                                                   }
{                                                                           }
{ IF Full_Expr-10 THEN { Loop_Statement } ...                               }
{            [ ELSE { Loop_Statement } ... ]   FI ;                         }
{ ;                                                                         }
```

Execute_Phase

Examples:

1. Specify an assignment that is executed once, at the beginning of a phase.

```
begin (accumulated_dollars := 0;)
```

2. Call for the printing of a line for each input record.

```
begin ( ... )
print a_detail_name_from_the_Detail_group;
```

3. Specify an assignment that is executed once per input record before any output line processing is done for the input record.

```
begin ( ... )
accumulated_dollars := accumulated_dollars + unit_dollars;
print the_detail_line;
```

4. Execute one of two sets of assignment statements with the selection dependent on the relationship between the value in an input field and a parameter.

```
dcl 1 parameter, 2 desired_color char(*);
dcl 1 input ...
     2 unit_color char(6), ...
 .
 .
 .
begin ( ... )
if unit_color = desired_color
   then color_match := color_match + 1;
        print color_line;
   else mismatch := mismatch + 1;
        print mismatch_line;
fi;
```

5. During the first pass through the input file, save the original input data and the current values of two local variables for each input record for use in a later phase.

```
begin ( ... )
hold input, color_match, color_mismatch;
```

6. Sort held file.

```
begin ( ... )
  .
  .
  .
hold input, unit_cost;
begin ( ... )
sort unit_cost descending;
```

Execute_Phase

7. As part of the third pass through the input information, save the current values of the input fields and the same local variables whose values were saved during the first phase.

```
begin ( ...  )
hold input, color_match, color_mismatch;
  .
  .
  .
begin ( ...  )
  .
  .
  .
begin ( ...  )
  .
  .
  .
hold;
```

8. Illustrate the sequence in which statements are executed.

```
dcl count decimal;
define 1 report ...
        2 detail parts_analysis,
          .
          .
          .
           4 part_number char(6) col 1
             let (count := count + 1;),
            .
            .
            .
begin (count := 0;)
     /* count = 0 */
count := count + 1;
     /* count = 1, 4, 7, ...  */
print parts_analysis;
     /* count = 2, 5, 8, ...  */
count := count + 1;
     /* count = 3, 6, 9, ...  */
end;
```

9. See the "hold_and_sort" example in Appendix B for an example that emphasizes the HOLD and SORT statements and their interaction with the PRINT statement.

Execute_Phase

10. Sorting example. Assume that the set of held records is as shown below, one record per line, and the column headings are the names of the input fields and/or local variables.

kind	stock	orders	price
apple	1	3	8
banana	2	5	10
cherry	1	3	9
fig	1	6	10
grape	2	5	7
lemon	1	6	7
melon	2	4	8
orange	2	4	8

When sorted with the statement:

```
sort stock asc, orders desc, price asc;
```

the ordering of the records becomes:

kind	stock	orders	price
lemon	1	6	7
fig	1	6	10
apple	1	3	8
cherry	1	3	9
grape	2	5	7
banana	2	5	10
orange	2	4	8
melon	2	4	8

Note that the melon and orange records are in a different order than they were in the input (see General Rule 12).

Syntax Rules:

1. The Format 1 subgroup must be used once and only once. It must appear immediately after the last Define_Report group.

2. The Format 2 subgroup may be omitted or may appear as many times as desired. The first occurrence of the Format 2 subgroup must be immediately after the only appearance of the Format 1 subgroup.

3. The last character of each occurrence of either of these subgroups is a semicolon. There may be several additional semicolons between the BEGIN keyword and the final semicolon.

4. In Format 1, the notation states that INPUT, input_field_name-4, and local_variable_name-3 can each occur an indefinite number of times. For example, the notation allows:

```
hold input, input, unit_cost, input, unit_cost;
```

to be supplied. MRPG accepts this statement. That is, the INPUT keyword and/or the name of an input field, or the name of a local variable can appear more than once. However, the redundant occurrences are ignored. Only one value is held in a record for each field or variable specified in the HOLD statement. The field and variable names and the INPUT keyword can occur in any order. The overall effect is independent of the order of occurrences.

5. If the BEGIN keyword occurs more than once in the program, more than one phase exists and a HOLD statement must be supplied in the first phase.

6. The PRINT!report_name-1 option can be used only if the report being called for defines exactly one DETAIL line.

7. The null form of a Loop_Statement is the semicolon character. Thus, the correct form of an IF statement that does nothing, should the test succeed, requires a semicolon between the THEN and ELSE keywords. For example:

```
if a = b then; else c := d; fi;
```

General Rules:

1. The BEGIN keyword constitutes the beginning of a phase. The first phase processes the original input records. All succeeding phases process the held records.

2. Assignment statements that exist within parentheses shown on the BEGIN line of each format diagram are executed once and only once. Execution is at the beginning of the phase (i.e., before the first record becomes available).

3. The expression specified for Full_Expr-9 is evaluated and the result is stored in the variable specified by local_variable_name-2.

4. Loop_Statements appearing after the closing parenthesis of the BEGIN clause and before the next BEGIN keyword (or the END keyword if this is the last phase) are executed once per record. The statements are executed in the order in which they occur in the source program. This set of statements constitutes the loop of statements that is executed once for each input record. A SORT statement is executed only once, near the beginning of a phase and before any statements in the Loop_Statement subgroup.

5. A HOLD statement is executed once per record (i.e., a HOLD statement is part of the loop). All held records have the same structure. The INPUT keyword in the HOLD statement keeps all input fields.

6. The HOLD statement in the first phase can cause none, any, or all of the input fields to be saved for later usage. If that HOLD statement includes any local_variable_name-3 entries, the current values of those local variables are saved as part of the held record. Thus, it is possible to save a different value of a local variable in each of the held records. See the "begin_hold_assign" example in Appendix B.

7. In the first phase, both of the following hold all of the input fields and nothing else:

```
hold input;
hold;
```

   Another way to hold all of the input and nothing else is to list the name of every input field following the HOLD keyword.

8. The sequence in which HOLD items are listed is not significant. The sequence in which the values are arranged in the held records is not significant because that sequence is not visible to the MRPG-OS.

9. The Loop_Statement subgroup can change the value associated with an input field, by having input_field_name-4 appear on the left side of an assignment statement. An assignment statement in the LET clause of the Report_Field_Def group can also change an input value. Should this happen, a subsequent HOLD statement that is executed in the same phase holds the new value. See the "begin_hold_assign" example in Appendix B.

10. The sequence in which actions occur is the sequence in which the statements triggering those actions occur within a phase. An assignment statement in a LET clause of the Report_Field_Def group is executed as part of a PRINT statement (see Example 9).

11. A SORT can be performed in the second and subsequent phases, but not on the original input records. However, the effect of sorting the original input records can be achieved by not modifying the values in any of the input fields before executing the HOLD statement in the first phase and saving the input fields in that HOLD statement.

12. The SORT keyword triggers the execution of a sort algorithm that is not guaranteed to preserve the input order of those records with identical values in the sort key fields (see Example 10).

13. If none of the ASCENDING or DESCENDING sorting direction keywords are supplied for a sort key name, the default is ASCENDING. Thus these are equivalent:

```
sort stock asc, orders desc, price asc;
sort stock, orders desc, price;
```

14. If a field or variable name is supplied more than once in the same SORT statement, the sorting direction used is the sorting direction associated with the last occurrence of the name.

15. Use of the NO DUPLICATE keyword results in discarding all but one of the records that have identical values in the sort key fields. In general, which one of the duplicate records is retained is unpredictable.

Execute_Phase

16. If a HOLD statement is used in the second and/or subsequent phases, the input fields and local variables saved are the same input fields and local variables that were saved by the HOLD in the first phase. The values in those fields' variables may all have been changed by assignment statements in the current and/or any intervening phases. See the "begin_hold_assign" example in Appendix B.

17. If the first phase includes a HOLD statement, it is valid for the next several phases to leave out the HOLD statement and for a later phase to include a HOLD statement. In this case, the same input is available to all those phases that lack a HOLD statement and to the first succeeding phase that includes a HOLD statement.

18. After the expression specified for Full_Expr-8 is evaluated, and if the identifier on the lefthand side of the assignment statement is a local_variable_name, then the Full_Expr-8 result is stored in that variable. If the identifier is an input_field_name, the Full_Expr-8 result is stored in the specified field in a temporary copy of the input file (i.e., nothing is stored by the MRPG-OS into the actual input file).

19. If exactly one DETAIL line was defined for a report and exactly one Detail group was defined for that same report, then the name that follows the PRINT keyword may be either the report name or the detail name. This is the only situation in which the PRINT report_name-1 option is valid.

20. A PRINT statement produces one set of detail lines. Usually there is only one line in the set. However, dependent upon conditional tests, the arguments supplied to the MRPG-OS, and the input data, the set of detail lines could yield no output, one line, or several printed lines.

21. If the current output line position is near the bottom of a page, the production of a set of detail lines can cause the production of page footing lines and page heading lines before the detail lines are written.

22. If the input record that triggered the new set of detail lines also caused a control break, detail footing and detail heading lines may be produced before the detail lines are written.

Execute_Phase

23. In the event that an IF statement appears, the value of Full_Expr-10 is tested and control is transferred within the IF statement as shown below. Control always reaches the "FI;" that denotes the end of the IF statement.

Test Result	Statements Follow THEN	ELSE Present	Statements Follow ELSE	Resulting Action
Succeeds	Yes			Execute statements following THEN. Then advance to the FI;
Succeeds	No			Advance to the FI;
Fails		Yes	Yes	Execute statements following ELSE. Then advance to the FI;
Fails		Yes	No	Advance to the FI;
Fails		No		Advance to the FI;

Nothing is actually done at the FI; point. In the above table, the remark "Advance to the FI;" means that the IF statement's execution is complete and control passes beyond the IF statement.

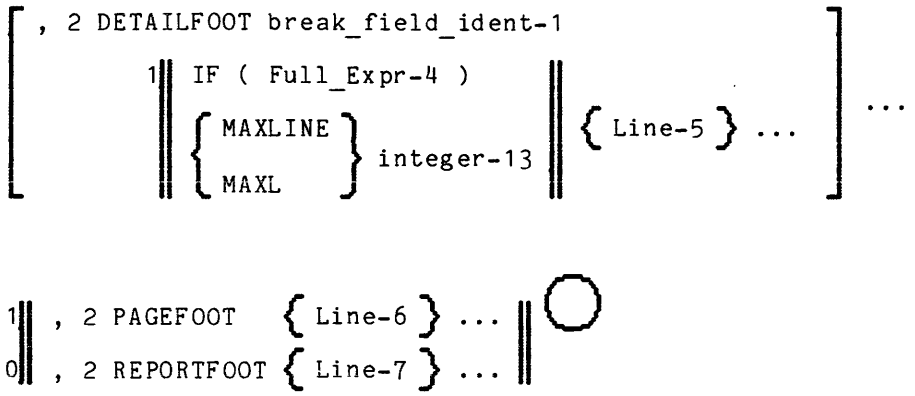
24. IF statements may be nested to a large, indeterminate depth. The limit is dependent on many characteristics of the program and cannot be described in detail for the general case.

25. A phase ends when the last statement in the loop is executed for the last input or held record. Any specified detail footing lines are produced. Control breaks occur at all defined levels and control break processing occurs. If specified, page footing and report footing lines are produced.

Group Name: Footing

The Footing group, and its subsidiary groups, specify the layout and content of detail footing lines, page footing lines, and report footing lines.

General Format:



Examples:

1. Print two lines at the bottom of every page. On the last page, print some text that applies to the report as a whole.

```
define 1 report payroll_deduction_analysis
                on file "your_output",
                .
                .
                .
              2 pagefoot,
                3 line 49, 4 "Fiscal 1977",
                3 line +1, 4 "Company Confidential",
              2 reportfoot,
                3 line 10, 4 "References:",
                3 line +2, 4 "1. Some text",
                3 line +1, 4 "continued",
                3 line +2, 4 "2.   More text",
                3 line +1, 4 "and still more.";
```

2. At execution time, select the detail footing line to print based on the department code number in the old record when the department code number changes between successive input records (i.e., a control break occurs).

```
              2 detailfoot dept_code if (dept_code = "123"),
                3 line, 4 "Totals for Welding Department",
              2 detailfoot dept_code if (dept_code = "124"),
                3 line, 4 "Totals for Grinding Department",
              2 detailfoot dept_code if (dept_code = "125"),
                3 line, 4 "Totals for Drilling Department",
```

Footing

Syntax Rules:

1. Either the PAGEFOOT or REPORTFOOT clause may be used first (i.e., if both are used).

2. The break_field_ident names must be declared as input_field_names in the Declare_Input_File group and be identified as control break fields in the BREAK phrase of the Report_Control group.

General Rules:

1. If the PAGEFOOT clause is used, the lines that it defines are produced at the bottom of every page.

2. If the REPORTFOOT clause is used, the lines that it defines are produced once, at the end of the report.

3. If both a REPORTFOOT clause and a PAGEFOOT clause are in a report's definition, the sequence of information on the last page is as follows. This sequence is independent of the order of the REPORTFOOT and PAGEFOOT clauses.

   - Some detail lines
   - The set of DETAIL lines for the last record
   - If specified, the DETAILFOOT lines for the last record
   - The REPORTFOOT lines
   - The PAGEFOOT lines
   - A newpage character

4. A DETAILFOOT clause defines a set of detail footing lines that may precede a set of detail lines when a control break occurs. If a DETAILFOOT clause is defined, and a control break occurs for the field specified in that DETAILFOOT clause, then the set of lines that are subservient to this DETAILFOOT clause is a candidate for being produced. There may be only one line in the set. When an IF test is specified, Full_Expr-4 is evaluated and the result is tested. Assuming that the IF test succeeds, the set of lines becomes a candidate for being produced. In the event that the IF test fails, the set of lines is omitted. Should no IF test be specified, the set of lines is a candidate for being produced. If the set of lines becomes a candidate for being produced, then a similar IF test within each line's definition may be specified to determine whether or not each line is to be produced. (See "Lines" group for details.)

5. This paragraph and the next three general rules occur with almost identical wording in the Heading and Detail groups. Small changes are made because the group name changes.

6. A MAXLINE value, known as report_maxline, is established for the report as a whole in the Report_Control group. If, in this Footing group, the MAXLINE integer-13 phrase is supplied, the value of integer-13 must be less than or equal to the value of report_maxline.

7. The line number of the highest-numbered line on which the first line of the DETAILFOOT set of lines may be printed is the minimum of report_maxline and integer-13, if the MAXLINE integer-13 phrase is supplied. If the MAXLINE integer-13 phrase is omitted, then report_maxline specifies the line number of the highest-numbered line on which the first line of this set is produced.

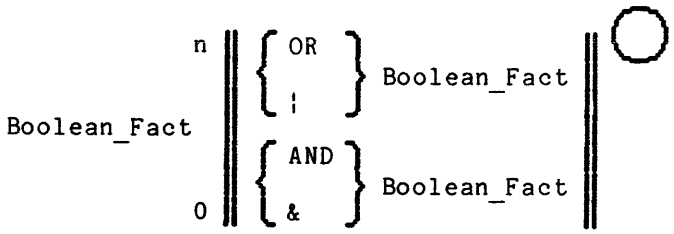
8. If more than one line is defined for the DETAILFOOT set of lines, then the value of integer-13 must not be larger than report_maxline minus the maximum quantity of lines that might be produced for the DETAILFOOT set being specified. As an example, suppose that the value of report_maxline is 45. Assume that the DETAILFOOT set of lines has six lines specified, but an IF test is specified for each line. Further assume that the user knows that the IF tests and the data in the input file are such that no more than four of the six lines are ever produced on the same page. Then, a value of 41 can be supplied for integer-13.

Full_Expr

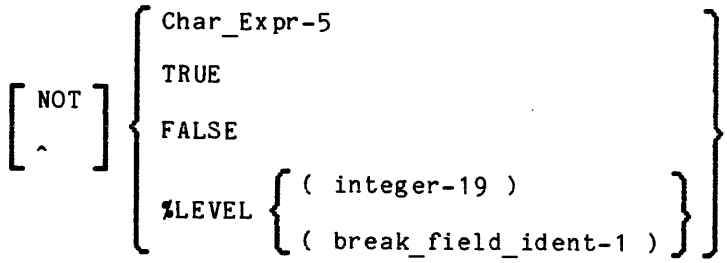
Group Name: Full_Expr

The Full_Expr group and its subsidiary groups provide a comprehensive arithmetic, character, and boolean expression capability. In addition to the details discussed here, Section 7 contains a unified treatment of the interactions between the rules stated in the Full_Expr group and its subsidiary groups.

General Format:



where Boolean_Fact is



Relationship_Test

Examples:

1. An example using the boolean operators OR, AND, NOT, and some relationship operators is:

```
salary > 1000 or pay_level > 9
name = Jones and ^married
```

2. Example relevant to control breaks.

```
declare 1 input ...
          2 part_class    char(10),
          2 vendor_name   char(20),
        .
        .
        .
define 1 report ...
          break (part_class, vendor_name)
        .
        .
        .
           2 detailfoot part_class ...
           2 detailfoot vendor_name ...
        .
        .
        .
if %level(2) then vendor_count := vendor_count + 1;
if %level(part_class) then print part_class_description;
```

3. Does the Pacific Ocean touch this state?

```
declare P_O_states set ("California", "Oregon",
        "Washington", "Alaska", "Hawaii");
                .
                .
                .
if state in P_O_states then salesman := "JA (Salty) Waters, Sr.";
```

4. In this example, "BF" stands for a Boolean Fact (i.e., an expression whose value is either true or false). The following sets are equivalent:

a.
```
BF_1 or  BF_2 and BF_3
BF_1 or (BF_2 and BF_3)
```

b.
```
BF_1 and   not  BF_2  and  not BF_3
BF_1 and ((not  BF_2) and (not BF_3))
BF_1 and   not (BF_2     or    BF_3)
```

c.
```
  BF_1 or  BF_2 and BF_3   or  BF_4 and BF_5
(BF_1 or (BF_2 and BF_3)) or (BF_4 and BF_5)
```

5. These examples illustrate the meanings of the boolean connectives. Assume that the following variables have the indicated values.

Variable	Value	Data Type
we_make_it	false	boolean
color	green	character
size	3	decimal

Expression	Result
we_make_it and size = 3	false
color = "green" or size = 4	true
color = "red" or not (size = 4)	true
^true	false
^we_make_it	true
not we_make_it	true

Full_Expr

Syntax Rules:

1. An indefinite number of terms may be strung together to form a "full expression".

2. The Full_Expr group and its subsidiary groups include a degree of recursiveness. Full_Expr references Char_Expr which references Arith_Expr which references Full_Expr, with the Full_Expr reference in Arith_Expr being enclosed in parentheses.

3. Char_Expr-5 and Char_Expr-6 may reference any type of parameters, input fields, built-in functions, and/or local variables.

4. The %LEVEL keyword may only appear inside the definition of a report, within the Heading, Detail, Footing, Line, and Report_Field_Def groups.

General Rules:

1. Boolean facts may be strung together with OR, AND, and NOT connectives. OR, AND, and NOT have their normal, logical meanings of union, intersection, and negation.

2. The order of expression evaluation is determined by the precedence of operators and by parenthesization. See Section 7 for a discussion of this topic for all operators.

3. The keywords TRUE and FALSE may use any mixture of uppercase and lowercase letters. For example, TRUE = true = TruE = True = tRUE.

4. %LEVEL is valid only when control breaks have been specified in the Report_Control group.

5. In %LEVEL (integer_19), the value of integer_19 is the number of the control break level of interest. The first field listed in the BREAK option of the Report_Control group is for level 1 breaks. The second field is for level 2 breaks, and so forth.

6. Assume that the current input record is record number 27, that record 27 did not cause a control break for level 3 or 2 or 1, that record 28 will cause a level 3 control break, but record 29 will not. %LEVEL (3) is true from the time that record 28 is available until the time that record 29 is available.

   In addition, %LEVEL (0) is true during the end-of-phase processing. Thus %LEVEL (0) is false until processing is completed for the last record.

7. The name of the break field may be used instead of the level number. For example, if a report definition contains:

   ```
   break (part_class, vendor_name)
   ```

   then the following are equivalent:

   ```
   $level (2)
   %level (vendor_name)
   ```

Full_Expr

8. The result of the IN clause is the value true or false.

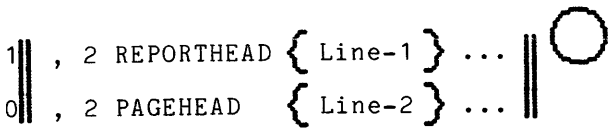
9. The result of the Relationship_Test is the value true or false.

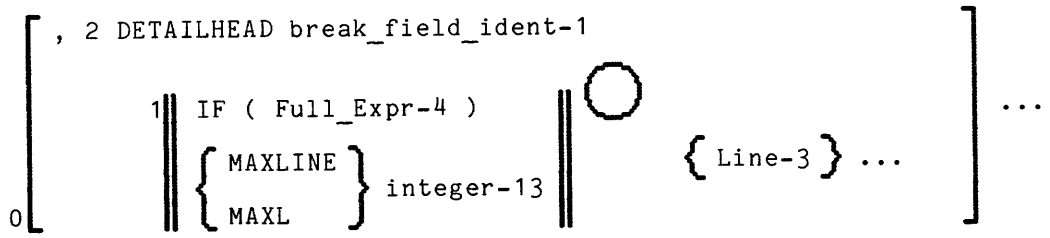
Heading

Group Name: Heading

The Heading group, and its subsidiary groups, specify the layout and content of report heading lines, page heading lines, and detail heading lines.

General Format:





Examples:

1. Print text on the cover page (e.g., a report heading). Then print two lines at the top of all following pages.

```
define 1 report payroll_deduction_analysis
                on file "your_output",
        2 reporthead,
          3 line 10, 4 "PAYROLL DEDUCTION ANALYSIS",
          3 line +2, 4 "1977 December 31",
          3 line +2, 4 "A summary of the ---",
          3 line +1, 4 "etc.  etc.  etc.",
        2 pagehead,
          3 line 1, 4 "PAYROLL DEDUCTIONS",
          3 line +1, 4 "Fiscal 1977",
```

2. Do not have a cover page. Put the report heading information on the same page as the first page of the body of the report.

```
        2 pagehead, ...
        2 reporthead, ...
```

3. At execution time, select the detail heading line to print based on the department code number in the new record when the department code number changes between successive input records (i.e. a control break occurs).

```
2 detailhead dept_code if (dept_code = "123"),
  3 line, 4 "Welding Department",
2 detailhead dept_code if (dept_code = "124"),
  3 line, 4 "Grinding Department",
2 detailhead dept_code if (dept_code = "125"),
  3 line, 4 "Drilling Department",
```

Syntax Rules:

1. Either the REPORTHEAD or the PAGEHEAD clause may be used first (i.e., if both are used).

2. The break_field_ident names must be declared as input_field_names in the Declare_Input_File group and be identified as control break fields in the BREAK phrase of the Report_Control group.

General Rules:

1. If the REPORTHEAD clause is used, the lines that it defines are produced once, at the beginning of the report.

2. If the PAGEHEAD clause is used, the lines that it defines are produced at the top of every page.

3. Assume that the REPORTHEAD clause and the PAGEHEAD clause are used in that order. The sequence of information starting at the beginning of the report is:

   - The REPORTHEAD lines
   - A newpage character
   - The PAGEHEAD lines (Page number is 1.)
   - If specified, the DETAILHEAD lines for the first record
   - The DETAIL lines for the first record
   - More DETAIL lines

4. Assume that the PAGEHEAD clause precedes the REPORTHEAD clause in the source program. The sequence of information on the first page is:

   - The PAGEHEAD lines
   - The REPORTHEAD lines
   - If specified, the DETAILHEAD lines for the first record
   - The DETAIL lines for the first record
   - More DETAIL lines

5. A DETAILHEAD clause defines a set of detail heading lines that may precede a set of detail lines when a control break occurs. If a DETAILHEAD clause is defined, and a control break occurs for the field specified in that DETAILHEAD clause, then the set of lines that are subservient to this DETAILHEAD clause is a candidate for being produced. There may be only one line in the set. When an IF test is specified, Full_Expr-4 is evaluated and the result is tested. Assuming that the IF test succeeds, the set of lines becomes a candidate for being produced. In the event that the IF test fails, the set of lines are omitted. Provided that no IF test is specified, the set of lines is a candidate for being produced. If the set of lines becomes a candidate for being produced, then a similar IF test within each line's definition may be specified to determine whether or not each line is to be produced. (See "Lines" group for details.)

6. This paragraph and the next three general rules occur with almost identical wording in the Detail and Footing groups. Small changes are made because the group name changes.

7. A MAXLINE value, known as report_maxline, is established for the report as a whole in the Report_Control group. If, in this Heading group, the MAXLINE integer-13 phrase is supplied, the value of integer-13 must be less than or equal to the value of report_maxline.

8. The line number of the highest-numbered line on which the first line of the DETAILHEAD set of lines may be printed is the minimum of report_maxline and integer-13, if the MAXLINE integer-13 phrase is supplied. If the MAXLINE integer-13 phrase is omitted, then report_maxline specifies the line number of the highest-numbered line on which the first line of this set is produced.

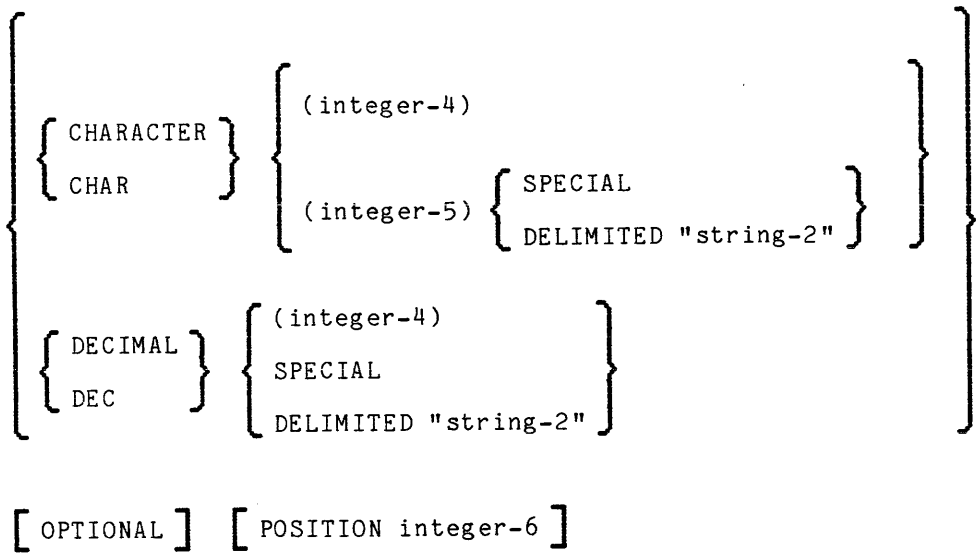
9. If more than one line is defined for the DETAILHEAD set of lines, then the value of integer-13 must not be larger than report_maxline minus the maximum quantity of lines that might be produced for the DETAILHEAD set being specified. As an example, suppose that the value of report_maxline is 45. Assume that the DETAILHEAD set of lines has six lines specified, but an IF test is specified for each line. Further assume that the user knows that the IF tests and the data in the input file are such that no more than four of the six lines are ever produced on the same page. Then, a value no larger than 41 can be supplied for integer-13.

Input_Field_Def

Group Name: Input_Field_Def

The Input_Field_Def group provides the detailed specification of one field in the input record.

General Format:



Examples:

These examples include a small part of the Declare_Input_File group in order to make each field definition complete. The initial "2", the field name, and the trailing comma or semicolon are all shown in the Declare_Input_File group.

1. Define a record with no gaps between the fields. In this sample record, 11111 represents the pay number and 2222222 represents the annual salary.

   Sample record: 111112222222Warren G. Wonka

   Definition:

```
2 pay_number       dec(5),
2 annual_salary    dec(7),
2 full_name        char(15);
```

2. The same record as in Example 1, but described using the POSITION phrase.

```
2 full_name        char(15) position 13,
2 annual_salary    dec(7)   position 6,
2 pay_number       dec(5)   position 1;
```

Input_Field_Def

There is no need to supply the field definitions in the reverse order shown, but it could be done this way.

3. Most people in the United States have three names (first, middle, last). Some people have more than three parts to their full name. Call these parts first, second, third, fourth, and last, to provide for up to five parts. To accommodate most people, let the third and fourth parts be optional. That is, the input records may or may not contain fields for the third and fourth names.

   Assume that the structure of the input records is as follows.

Field No.	Field Name	Is The Field Length Fixed or Variable?	Minimum Field Length	Maximum Field Length
1	pay_number	Fixed	4	4
2	annual_salary	Fixed	5	5
3	last_name	Variable	1	20
4	first_name	Variable	1	19
5	second_name	Variable	1	18
6	third_name	Variable	0	17
7	fourth_name	Variable	0	16

   Fields 6 and 7 may or may not be present. If a person has no middle name or initial, field 5 contains the single character "9".

   The last field (5, 6, or 7) is followed by a newline character, denoted by <NL> below. Fields 3 through the last field are separated by commas.

   Several examples of input records are shown. In these examples, the pay number is represented by four odd-value digits (e.g., 1111, 3333). The annual salary is represented by five even-valued digits (e.g., 22222, 44444).

```
111122222Edgerton,Jonathan,Micheal<NL>
333344444Engels,Albert,Bertram,Charles,David<NL>
555566666Green,Marybelle,9<NL>
777788888Brown,Horace,Smedley,Eustice<NL>
```

   A valid declaration for this file is:

```
dcl 1 input stream file "your_input",
    2 pay_number        dec(4),
    2 annual_salary     dec(5),
    2 last_name         char(20) delimited ",",
    2 first_name        char(19) delimited ",",
    2 second_name       char(18) delimited ",",
    2 third_name        char(17) delimited "," optional,
    2 fourth_name       char(16) optional;
```

Input_Field_Def

Syntax Rules:

1. The Input_Field_Def group is used once for each field that is declared in the Declare_Input_File group except for those portions of the input record that are declared with the FILL option in the Declare_Input_File group.

2. All fields that include the OPTIONAL keyword in their declaration must reside adjacent to each other in the input record and must be at the end of the record.

General Rules:

1. The CHARACTER keyword causes the input field to be treated as a character string.

   a. If the CHARACTER (integer-4) combination is selected, exactly integer-4 characters are taken from the input. The internal version of the input field has a length of integer-4 characters.

   b. If the CHARACTER (integer-5) combination is selected, the quantity of characters taken from the input is determined by either the SPECIAL or the DELIMITED mechanism, as discussed in later rules. The internal version of the input field has a maximum length of integer-5 characters, but it is valid for the quantity of characters taken from the input field to be greater than integer-5. If this occurs, only the first integer-5 characters are available to the program. However, the current position in the input data advances further, to the point specified by the SPECIAL or the DELIMITED mechanism.

2. The DECIMAL keyword causes the input field to be treated as a decimal value.

   a. The internal representation is float decimal(20). This float decimal(20) representation continues to be used even if the input field becomes a held field by means of the HOLD keyword in the Execute_Phase group.

   b. If the DECIMAL (integer-4) combination is selected, exactly integer-4 characters are taken from the input. A newline character does not terminate an input field.

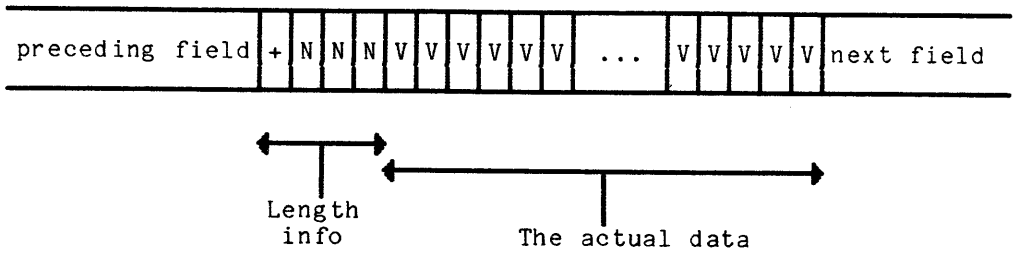
   c. If the DECIMAL SPECIAL combination or the DECIMAL DELIMITED "string-2" combination is selected, the quantity of characters taken from the input is determined by the SPECIAL or the DELIMITED mechanism, as discussed in other rules.

   d. The characters taken from the input (however selected) undergo a normal character to decimal conversion, as described in Section 6. Leading and/or trailing space characters are ignored when the string of input characters are converted to their internal decimal value. This is a PL/I characteristic.

Input_Field_Def

3. SPECIAL means that the actual, current length of the field is in the input file along with the current value of the field. If the input file is obtained from LINUS with a report request, then the SPECIAL keyword must be part of the definition of every field.

   The length and value of a SPECIAL field are arranged in the following manner in the input records, where each small box represents one character:



   where:

   NNN is the quantity of characters contained in the field's value.

   VV...VV is the field's value.

   The length data always occupies four characters, the first of which is always a plus character. The data format is fixed decimal(3).

   For example:

```
... preceding field+016A SAMPLE STRING.next field...
... preceding field+001Xnext field...
... preceding field+000next field...
```

4. If SPECIAL is used for two fields in succession, and there is an inconsistency between the length value for the first field and the actual length of that field, it is highly probable that an Illegal Procedure condition will occur when the MRPG-OS attempts to use the second field's length value. Consider the example:

```
some data+006EAGLE+005SCOUTmore data
```

   The +006 value should be +005 to match the length of the EAGLE string. With the +006, the characters EAGLE+ are used for the first field. The next four characters, "005S", are used as the length value for the next field. But when the 005S is interpreted as a fixed decimal(3) value by the hardware, an Illegal Procedure condition is detected and signalled. This happens because the hardware requires that the first character of a fixed decimal value must be either a + or a - character. Since the hardware found a zero, which is illegal, the Illegal Procedure condition is detected. Whether or not the three characters that specify the length of the data are valid or invalid depends on the four low-order bits of each of the three characters in the length field. If the value of the four low-order bits is greater than nine, a hardware-detected fault occurs.

   Although the hardware allows the sign character to be either a + or - for the length field, only the + character is acceptable to MRPG.

Input_Field_Def

5. When DELIMITED is chosen, string-2 specifies the single character that immediately follows the field being defined. If the input file is a stream file, then the string-2 character can be a newline character to signify the end of the last field in a record.

6. The value of integer-6 in a POSITION phrase specifies the character position of the first character of the field being defined. Conventional numbering is used (i.e., the first character of the record is in character position 1).

Group Name: Line

The Line group enables the user to specify where the line is to be placed on the printed page, whether or not the line is to be produced, and whether or not the MRPG-OS should pause to allow a terminal operator to place a new, blank form in the terminal.

General Format:

```
, 3 LINE { [ integer-14 ]                                                    }
         { [ + integer-15 ] [ IF ( Full_Expr-5 ) ] [ Report_Field_Def ] ... }
         {
         { { PAUSE } [ IF ( Full_Expr-6 ) ]                                  }
         { { 0     }                                                         }
```

Examples:

1. Specify a line that is to appear on line 6 of a page if a condition is satisfied.

```
3 line 6 if (dept_code = 123) ...
```

2. Specify a line that is to appear on the next line.

```
3 line +1, 4 ...
```

-OR-

```
3 line, 4 ...
```

3. Close out the current page and position to the end of the current page.

```
3 line 0,
```

4. Specify a line that enables the user to remove a completed form and insert a blank form in the terminal.

```
3 line pause,
```

5. Specify that a blank line is to be produced between two non-blank lines.

```
3 line,     4 some_field char (20),
3 line +2,  4 another_field char (20),
```

Another way:

```
3 line, 4 some_field char (20),
3 line,
3 line, 4 another_field char (20),
```

Syntax Rules:

1. Examination of the Heading, Detail, and Footing groups shows the Line group may be used several times in a row to define a set of lines that are treated as a unit.

2. The option grouped with the PAUSE option is the digit zero.

3. The difference between the integer-14 and the integer-15 choices is the plus sign and the subsequent interpretation.

4. There are two possible major choices within the large braces. The first major choice is comprised of the three sets of brackets at the top of the general format. Since all three items are in brackets, and hence optional, it is possible and valid for one major choice to be nothing. The second major choice is comprised of the small braces containing PAUSE and 0 followed by the optional IF test.

General Rules:

1. If the integer-14 choice is made, it specifies the absolute line number that the line is to be printed on. The minimum value of integer-14 is one.

2. If the "+ integer-15" choice is made, it specifies where the line is to be printed relative to the previous line. The value +1 results in single spacing, +2 is double spacing (i.e, one blank line between printed lines). The value +0 should not be used (i.e., overprinting cannot be handled).

3. If the upper portion of the general format is chosen and neither integer-14 nor integer-15 are specified, the default produces single spacing just as though +1 is specified.

4. When an IF (Full_Expr-5) test is specified, Full_Expr-5 is evaluated and the result is tested. In the event that the IF test succeeds, the line is produced, with a newline character as the last character of the line. Should the IF test fail, the line is not produced nor is a newline character produced. If the line becomes a candidate for being produced, then a similar IF test within each field's definition may be specified to determine whether or not each field is to be produced. (See "Report_Field_Def" group for details.) When the line is produced, a newline character is written at the end of the line, even if some fields are defined but none of them are produced, or even if no fields are defined.

5. It is possible to define one or more fields and omit the IF (Full_Expr-5) test. In this case, a newline character is written at the end of the line, whether or not anything is produced for any of the fields.

6. It is possible to select as the major path in the Line group the upper portion of the general format, but to omit all three items shown in brackets by using:

```
      .
      .
      .
   3 line,
      .
      .
      .
```

If this is done, a line that consists of only a newline character is produced, giving a blank line in the output.

7. When an IF (Full_Expr-6) test is specified, Full_Expr-6 is evaluated and the result is tested. In the event that the IF test succeeds, the actions described below for the PAUSE keyword or 0 that preceded the IF test occur. Should the IF test fail, then the actions do not occur.

8. If a zero digit follows the LINE keyword and there is either no accompanying IF test, or an accompanying IF test succeeds, the following occurs:

   - The page is closed out (i.e., any specified page footing lines are produced)
   - The output advances to an end of page position

9. If the PAUSE keyword follows the LINE keyword and there is either no accompanying IF test, or an accompanying IF test succeeds, the following occurs:

   - The two steps described in General Rule 8
   - If the output is being sent to the user_output I/O switch the MRPG-OS pauses until a newline character is read from the user_input I/O switch

Usually, user_input and user_output are attached through the tty_ I/O module to the user's terminal. This enables the user to print a page of the report on a high-print-quality terminal and have the MRPG-OS pause so that the user can remove that page from the terminal, insert and position a fresh sheet of paper, and then direct the MRPG-OS to continue producing the report by pressing the RETURN key. With this technique, the user can produce a report with a carbon film ribbon on a preprinted form. However, the actions listed above occur even if user_output is not attached to a terminal (e.g., user_output being sent via vfile_ to a segment). But in this case, it is difficult for the user to know when the MRPG-OS pauses. Similarly, user_input can be attached to a file rather than a terminal, but this complicates knowing when to provide the newline characters from user_input. Therefore, it is recommended that user_input and user_output be attached to a terminal whenever the PAUSE feature is used.

10. It is possible for one occurrence of the Line keyword to produce more than one output line. This can happen if either the FILL or the FOLD option of the Report_Field_Def group is specified. The quantity of output lines can vary in the same report, depending on the input data or on other variables. (See "FILL" and "FOLD" examples in the Report_Field_Def group).

11. If the LINE integer-14 option is chosen more than once within the declaration of a report, the integer-14 values must increase as those values are encountered in progressing through the report's definition.

12. There is one circumstance in which the quantity of extra blank lines is less than the quantity specified by integer-15. This occurs when the current line of the output report is near the bottom of a page and a "3 LINE +integer-15" phrase is encountered. Suppose that the LINE phrase of interest is part of a DETAIL set of lines. Further suppose that there is room for only three more detail lines on the current page, and a 10 was specified for integer-15. Three blank lines are appended to the report. If specified, page footing and page heading lines are also appended, but the remaining seven potential blank lines are omitted.

    A specific example can be seen in Section 2 (refer to Figure 2-3). Line 29 of the MRPG source program is "3 line +3," and is intended to cause two blank lines to precede the DETAILHEAD line. This occurs at lines 17-20 and 30-33 of the output report. However, although there are two blank lines for lines 5-6 of the report, they are not caused by line 29 of the source program. The +3 on line 29 is ignored, inasmuch as the report is initially at the top of a page. Rather, lines 5-6 of the report are blank because of the +2 on line 27 of the source program. Line 27 is part of the PAGEHEADING set of lines.

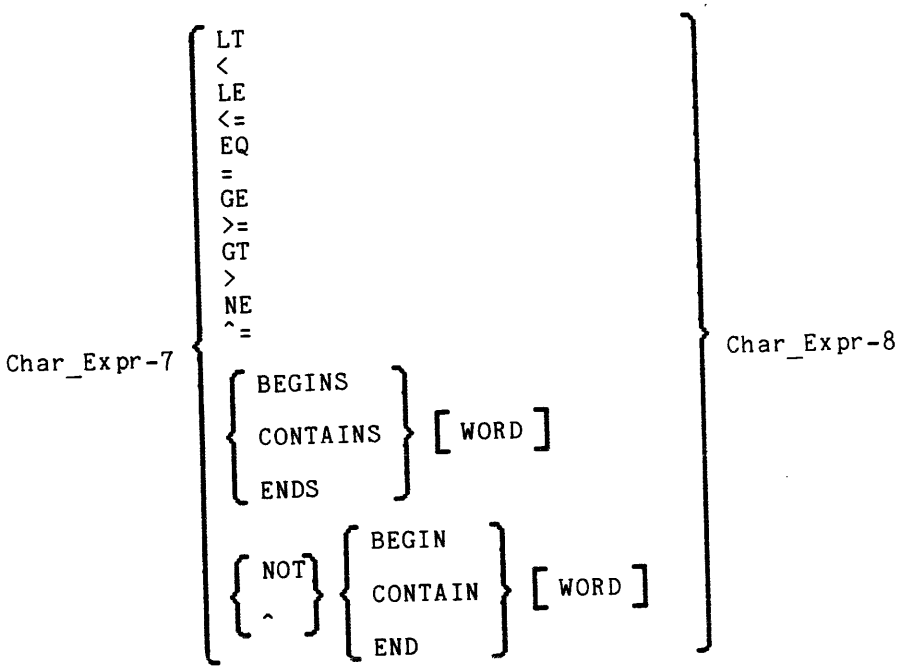
    The general rule is that no "extra" lines are produced at the top of a page.

Relationship_Test

Group Name: Relationship_Test

The Relationship_Test group is logically part of the Full_Expr group, and is separated only because it is awkward putting both groups on the same page because of space requirements. In addition to the details discussed here, Section 7 contains a unified treatment of the interactions between the rules stated in the Full_Expr group and its subsidiary groups.

General Format:



Examples:

1. Numeric example.

```
salary > 10000
tax_rate = 4.5
tax_rate eq 4.5
part_cost <= cost_target
```

2. Character string example.

```
job_title ^= "Manager"
"Jones" = last_name
```

Relationship_Test

3. String matching example. Assume that four variables contain the following and that each white space area between the quotation marks contains one or more space characters.

```
title_1  "fundamentals of geometry"
title_2  "builtin functions"
title_3  "only for fun"
title_4  "  fun  can  be  fun  "
```

This expression			has this value
title_1	begins	"fun"	true
title_1	begins	"Fun"	false
title_1 not	begin	"fun"	false
title_1	contains	"fun"	true
title_2 not	begin	"fun"	true
title_2	ends	"fun"	false
title_2 not	end	"fun"	true
title_2	contains	"fun"	true
title_3	ends	"fun"	true
title_4	begins	"fun"	false
title_4	contains	"fun"	true
title_4	ends	"fun"	false

4. This example is identical to the above except that the matching is done on a word basis, rather than on a string basis.

```
title_1 "fundamentals of geometry"
title_2 "builtin functions"
title_3 "only for fun"
title_4  "  fun  can  be  fun  "
```

This expression				has this value
title_1	begins	word	"fun"	false
title_1	begins	word	"Fun"	false
title_1 not	begin	word	"fun"	true
title_1	contains	word	"fun"	false
title_2 not	begin	word	"fun"	true
title_2	ends	word	"fun"	false
title_2 not	end	word	"fun"	true
title_2	contains	word	"fun"	false
title_3	contains	word	"fun"	true
title_4	begins	word	"fun"	true
title_4	ends	word	"fun"	true

Relationship_Test

Syntax Rule:

There are no constraints on the types of expressions whose relationship may be tested. Char_Expr-7 and Char_Expr-8 may be decimal, character, or boolean values. Thus, there are nine data type matching and mismatching combinations. The general rules in this group specify which value is converted into a different type for the six mismatching combinations. Section 6 provides the detailed conversion rules for the possible conversions.

General Rules:

1. The alphabetical and mathematical notation used for the six mathematical operators are:

LT	<	less than
LE	<=	less than or equals (not more than)
EQ	=	equals
GE	>=	equal to or greater than (not less than)
GT	>	greater than
NE	^=	not equal

2. To perform a relationship test, both operands must be of the same data type. When necessary, MRPG generates the PL/I statements needed to convert Char_Expr-7 or Char_Expr-8.

3. If the relationship operator is one of the mathematical operators (see General Rule 1) and both of the operands are of the same type, then no conversion is needed. The comparison is made and the result is either true or false. If the data types of the operands differ, the following table specifies which data type is converted to a temporary value.

Data Type Combinations	Data Type Conversion
boolean and decimal	boolean to decimal
boolean and character	boolean to character
decimal and character	decimal to character

4. The string operators are listed in this rule and explained in later rules.

BEGINS	CONTAINS	ENDS
NOT BEGIN	NOT CONTAIN	NOT END

5. If the relationship operator is one of the string operators listed above, then Char_Expr-7 and Char_Expr-8 are both converted to character values if they are not already character values.

6. When two character expressions of unequal length are compared, the shorter expression is assumed to have sufficient trailing spaces to make the lengths equal.

7. The BEGINS, CONTAINS, and ENDS operators determine if Char_Expr-8 appears in the specified position within Char_Expr-7. For the BEGINS test to be satisfied (i.e., give a "true" result) the character string specified by Char_Expr-8 must occur at the beginning of the string specified by Char_Expr-7. For the ENDS test to be satisfied, the

Relationship_Test

Char_Expr-8 string must occur at the end of the Char_Expr-7 string. For the CONTAINS test to be satisfied, the Char_Expr-8 string must occur somewhere in the Char_Expr-7 string, including at the start or the end of the Char_Expr-7 string. For the NOT versions of the tests to be true, the Char_Expr-8 string must not occur in the specified position.

8. When the WORD keyword is omitted, searching is done on a strict character by character basis. Char_Expr-8 can contain white-space characters. If Char_Expr-8 has three adjacent spaces between non-space characters, then in order for the result to be true, Char_Expr-7 must have exactly three spaces between the surrounding, matching, non-space characters. (See Example 3 .) Contrast this with the next rule.

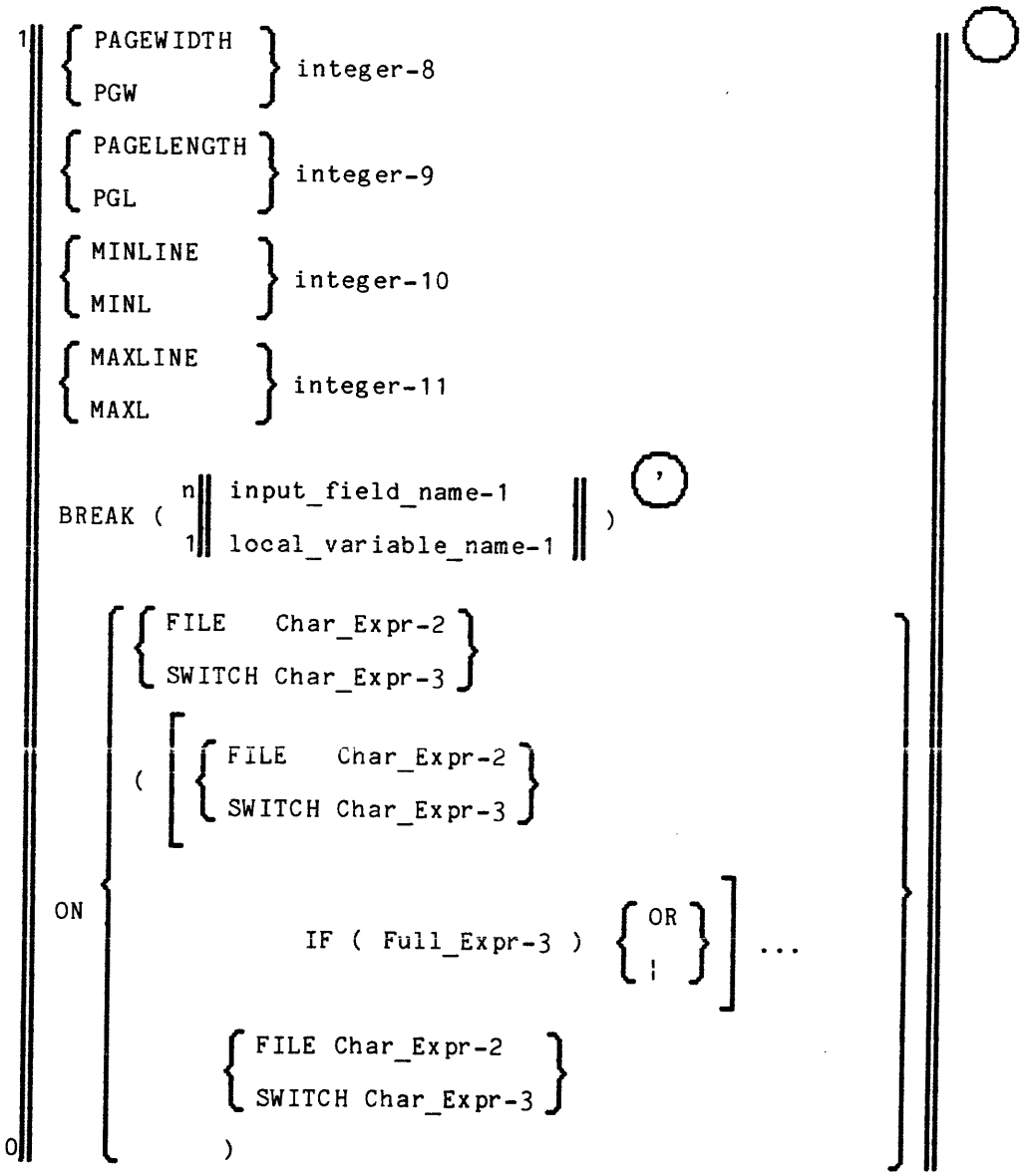
9. When the WORD keyword is specified, searching is done on an English word basis. Char_Expr-7 can be thought of as being subdivided into substrings by the Char_Expr-7 delimiters. The ends of Char_Expr-7 are delimiter positions. White-space characters within Char_Expr-7 are delimiters. One or more contiguous white-space characters are treated as one delimiter. If Char_Expr-8 contains any white-space characters, the result is false, independent of the value of the Char_Expr-7 string. (See Example 4.)

Report_Control

Group Name: Report_Control

The Report_Control group enables the user to specify several values and actions that apply to the report as a whole.

General Format:



Report_Control

Examples:

1. Specify the amount of space on a page that is available for the report.

```
pagewidth 70
pagelength 50
minline 5 /* This is the range of */
maxline 45 /* lines for detail info.  */
```

2. Identify input fields whose change of value triggers a control break.

```
break (department_code, pay_class)
```

3. Specify the segment into which the report is written.

```
on file "your_output"
```

Another way to send the report to the same segment is:

```
on switch "vfile_ your_output"
```

4. Specify that the segment into which the report is written depends on the value of a parameter that is received when the MRPG-OS is invoked.

```
dcl 1 parameter, 2 where char(*), 2 o_f char(*)
       Key ("-of") DEFAULT "";
 .
 .
 .
define 1 report payroll_analysis
       on (file "pay_anal.exempt" if (where = "e")
          or file "pay_anal.nonexempt" if (where = "n")
          or file "pay_anal.hourly")
 .
 .
 .
define 1 report latent
        on (file "output" if (o_f ^="")
           or switch "user_output")...
```

Syntax Rules:

1. Notice in the general format that the digit "1" at the upper left of the left-hand double bars indicates the highest-level options can occur only once.

2. A name should not appear more than once in the list for a BREAK option. The same name may be used in BREAK options in different reports. Repetition may occur within the ON option, as indicated by the ellipsis.

Report_Control

General Rules:

1. The value of integer-8 specifies the maximum quantity of horizontal printing positions (i.e., print columns). The default value is no limit. If the default is used and a report line does not fit on the output device, the normal Multics standards govern the handling of the excess characters, which are usually continued onto the next line. The PAGEWIDTH value is the same for every line in the report.

2. The value of integer-9 specifies the quantity of lines on the physical sheet of paper. There are some subtleties here concerning exactly how the report can be printed that are deferred to Section 9 of this manual. The default value is 66 if the report is being written to a segment. The PAGELENGTH value is the same for every page of the report. If the report is being written on user_output, the default value is no limit.

3. The value of integer-10 specifies the number of the lowest-numbered line on which a detail heading, detail data, or detail footing line can appear. If the page and/or report heading lines occupy fewer than one less than integer-10 lines, enough empty lines are skipped so that the first detail type of line falls on the integer-10th line.

4. If the MINLINE option is not specified, and no page heading lines are defined, the default value for MINLINE depends on where the report is to be written. (The FILE and SWITCH keywords may occur several times, but the conditional tests of Full_Expr-3 result in either one SWITCH option or one FILE option being selected at execution time.)

   a. If a FILE option is selected, the default MINLINE value is 4.

   b. If a SWITCH option is selected, and the name of the I/O switch is user_output, the default value of MINLINE is 1.

   c. If a FILE or SWITCH option is not specified, the default value for MINLINE is 1.

5. If the MINLINE option is not specified, but page heading lines are defined, the default value for MINLINE is one more than the quantity of page heading lines that are defined (i.e., MINLINE is one more than the quantity of times that the LINES keyword occurs subservient to the PAGEHEAD keyword). This means that it is possible to define six page heading lines and also know that no more than four will ever appear on a page because of the conditions chosen and the nature of the input data. If this is done, the default MINLINE value is 7, but it is valid to include MINLINE 5 in the report definition, which is the only way to utilize the two extra lines.

6. However arrived at, some value is established at generation time for the MINLINE value for the report as a whole. For use in the discussion of other groups, call this MINLINE value report_minline. This value is established even if the entire Report_Control group is omitted.

7. The value of integer-11 specifies the number of the highest-numbered line on which a heading, detail, or footing line may appear. If line integer-11 is reached and another detail line becomes available for output, the old page's footing lines are written, an advance is made to the next page, the new page's heading lines are written, and then the detail line is written.

8. If the MAXLINE option is not specified, but page footing lines are defined, the default value for MAXLINE is one less than the absolute line number of the first page footing line. (See "PAGEFOOT" in the Footing group and "integer-14" in the Line group.)

9. If the MAXLINE option is omitted and page footing lines are undefined, the default value for MAXLINE is the number of the last line on the page.

10. However arrived at, some value is established at generation time for the MAXLINE value for the report as a whole. For use in the discussion of other groups, call this MAXLINE value report_maxline. (The value is established even if the entire Report_Control group is omitted.)

11. Names listed in a BREAK option specify input fields or local variables whose change in value from one record to the next triggers a control break. The names are used in DETAILHEAD options in the Heading group and in DETAILFOOT options in the Footing group. The first name in the list is associated with break level 1, the second with level 2, etc. These level numbers are used with the %LEVEL option in the Full_Expr group. When a break occurs, that level and all higher-numbered levels also have a break.

    For example:

    ```
    break (dept_code, pay_class, job_title)
    ```

    When a new record is obtained, if the content of the pay_class field changes, then a level 2 break occurs. In addition, a level 3 break for job_title also occurs, whether or not the content of the job_title field changed. The detail footing lines for level 3 are produced, followed by the detail footing lines for level 2, then the detail heading lines for level 2, followed by the detail heading lines for level 3, and finally, the detail lines associated with the input record that triggered the control break.

    The test that determines whether or not a control break occurs is made near the beginning of the execution of the PL/I statements that are generated for a PRINT statement. These statements are executed for each input record. If a control break occurs, the current value of the input field or local variable is saved for use in the test for a later record. Should the control break be at other than the highest-numbered level, the input fields and/or local variables for this level and for all higher-numbered levels are saved.

    Usually, input fields are used for control breaks. A local variable whose value is saved for each input record has the general appearance of an input field in subsequent phases. A HOLD statement is used to save the values that the input fields and local variables have for each input record. The local variable named in a BREAK option need not have had its values saved by a HOLD statement. However, unless the value of that local variable is changed by assignment statements in LET options of the Report_Field_Def group, no control breaks occur for this local variable.

    The maximum quantity of break levels is 998.

12. The ON option specifies where the report is to be written. If the ON option is omitted, the report is sent to the user_output I/O switch.

Report_Control

13. Char_Expr-2 must be a character string. It is used by the vfile_ I/O module as the relative or absolute pathname of a segment for the file. The MRPG language does not impose any constraints on the characters in the string. However, the intended use of this string does impose constraints. Allowable characters and the length of the string are restricted to what is allowed in relative pathnames of segments. (See "Section 3" of the MPM Reference Guide.)

14. Char_Expr-3 is a character string that describes the destination. If the string contains no blanks, it is assumed to be a switch name. Otherwise, it is used as an attach description for an I/O module, usually the vfile_ module. The ATTACH keyword must be supplied to use an I/O module other than the vfile_ I/O module. (Refer to the MPM Subroutines, MPM Subsystem Writers' Guide, or the MPM I/O manuals for details of the required attach description.)

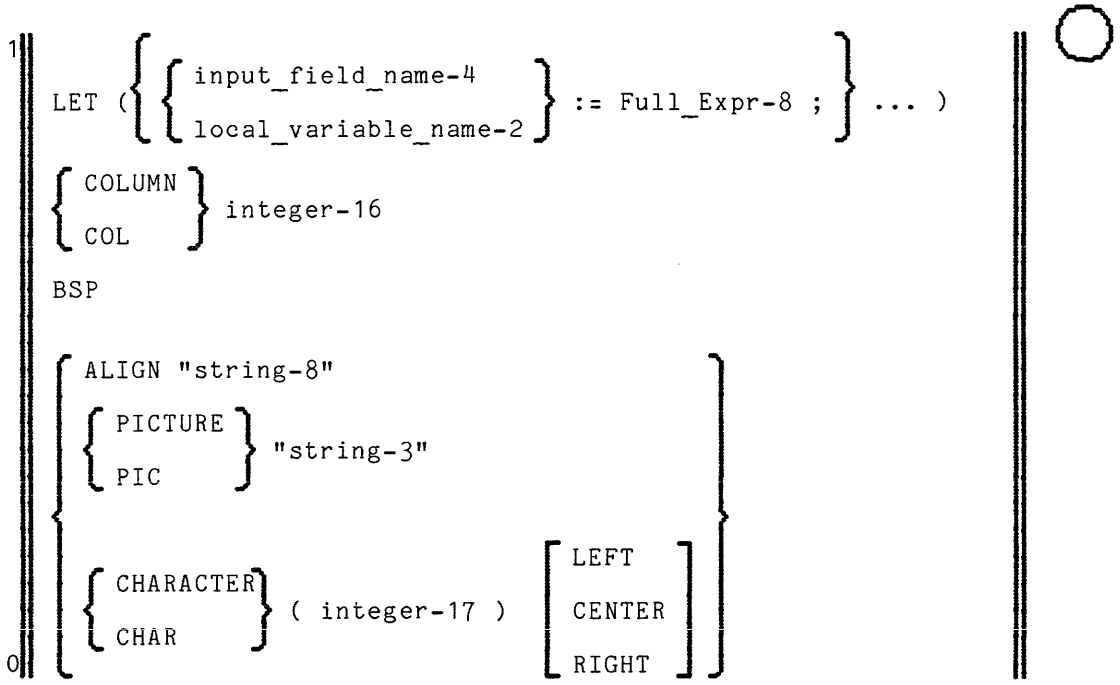
15. The IF test permits the report to be sent to different places, depending on the result of evaluating Full_Expr-3. Keep in mind that the Char_Expr-2 or Char_Expr-3 which is used belongs to the FILE or SWITCH option that precedes the IF test. The first IF test that is satisfied ends the output path selection process. Assuming that none of the IF tests are satisfied, the report is sent to the destination specified by the last FILE or SWITCH phrase (i.e., the FILE or SWITCH that immediately precedes the right parenthesis that ends the ON clause).

Report_Field_Def

Group Name: Report_Field_Def

The Report_Field_Def group defines the content, format, and position within a line of one field to be printed.

General Format:



Examples:

Several examples include items from the Char_Expr group to illustrate the interaction between certain character expressions and the keywords specified in this Report_Field_Def group.

Report_Field_Def

1. Define a line that produces the fourth through sixth lines shown below. The first line identifies the fields. The second line contains the starting column positions of each field. The third line indicates where characters can fall in the field.

```
pay_no dept salary      last_name      degree
1      8    13          24             38
xxxxx  xxx  xxxxxxxxx   xxxxxxxxxxxx   xxxx
11111  275  $2,567.00   Anderson        MBA
22222  349    $487.00   Lewis
33333  583    $969.72   Bradford         BS
```

```
4 pay_no     char (5),
4 dept       char (3) col 8,
4 salary     picture "$$,$$9v.99" col 13,
4 degree     char (4) right col 38,
4 last_name  col 24 char (12) left,
```

2. Define a field that is produced only if the department code is greater than a value supplied as a parameter.

```
dcl 1 parameter, 2 dept_code_limit;
 .
 .
 .
4 if (dept_code > dept_code_limit) "Invalid dept code" col 70,
```

3. LET example. Assume that the MRPG-OS finished processing the 27th input record and produced an output line. Further assume that:

```
The input field "cost" of record 28 contains 300
The input field "cost" of record 29 contains 410
The local variable "factor" contains 10.
```

Then:

```
4 cost + factor let(cost := cost + 30; factor := factor + 5;)
```

```
yields   345 for record 28 ((300 + 30) + (10 + 5) => 345)
and      460 for record 29 ((410 + 30) + (15 + 5) => 460).
```

After processing record 29, factor contains value 20.

Report_Field_Def

4. LET clause execution. Assume that:

   - Record 35 has just been processed
   - The input field "weight" of record 36 contains 157
   - The local variable "shrinkage" contains 23

   Now consider the following:

```
4 if (weight < 150) weight - shrinkage
   let (weight := weight - 10;
        shrinkage := shrinkage + 1;),
```

   After the assignments in the LET statements are executed, weight contains 147. Then the IF test is made. Since 147 is less than 150, the IF test succeeds. The evaluation of this report field yields 123 in the output line, which is calculated as (157 - 10) - (23 + 1). Weight now contains 147 and shrinkage now contains 24.

5. An example using BSP (backspace). Assume the field title contains:

   PART COST

   The string is 25 characters long (it contains eight letters, eight backspaces, and nine underscores). It can be printed using the following field definition:

```
4 title char (9) bsp,
```

6. ALIGN example. Assume that the test_data field in successive input records contains the following information. (The decimal points are actually present in the input fields.)

```
1.234Ъ
57.9ЪЪ
389.72
Ъ87.ЪЪ
ЪЪЪЪ54
```

   Then:

```
 4 test_data col 21 align ".",
```

   yields:

```
  1.234
 57.9
389.72
 87.
 54
```

   with the decimal points in column 21. The last line, not having a decimal point in the input field, is positioned with the 4 in column 20, just as though a decimal point had been present in the input immediately following the "4" character.

Report_Field_Def

7. OVERLAYING of data. Assume the heat_factor field in the next input record contains 123.456 and that some of the fields for an output line are defined as:

```
4 "abcde" col 21,
4 "+" col 26,
4 "fghij" col 27,
```

The output result is:

```
abcde+fghij
```

Replacing the second field to give:

```
4 "abcde" col 21,
4 heat_factor col 26 align ".",
4 "fghij" col 27,
```

yields:

```
ab123.fghij
```

Interchanging the second and third fields to give:

```
4 "abcde" col 21,
4 "fghij" col 27,
4 heat_factor col 26 align ".",
```

yields:

```
ab123.456ij
```

8. The physical placement of characters in a printed line can be affected by an IF test and the content of the previous field. The information arrangement is:

line from MRPG program

content and length of field_2 yields printed result

a. A base case:

3 line, 4 "FIRST_" col 1 char(6), 4 field_2 char(5), 4 "-THIRD"

field 2	length	yields
piggyback	9	FIRST_piggy-THIRD
piggy	5	FIRST_piggy-THIRD
pig	3	FIRST_pigbb-THIRD
	0	FIRST_bbbbb-THIRD

In the last line, field_2 contains nothing (i.e., a null character string).

Report_Field_Def

b. Omit the char(5) for field_2:

```
3 line, 4 "FIRST_" col 1 char(6), 4 field_2, 4 "-THIRD"
```

field_2	length	yields
piggyback	9	FIRST_piggyback-THIRD
piggy	5	FIRST_piggy-THIRD
pig	3	FIRST_pig␣␣-THIRD
	0	FIRST_␣␣␣␣␣-THIRD

c. Specify an IF test that results in field_2 being used. Assume that test_field has 5 as its current value.

```
4 if (test_field = 5) field_2 ...
```

gives the same results as in Examples a and b.

d. Specify an IF test that fails and do not specify a position for the following field:

```
3 line,
 4 "FIRST_" col 1 char (6),
 4 if (5 = 7) field_2 char (5),
 4 "-THIRD",
```

yields the result FIRST_-THIRD which is independent of the length and content of field_2 and independent of whether or not a COLUMN or CHARACTER option is specified for field_2.

e. Change "Example d" by specifying a column position for the field that follows field_2:

```
3 line,
 4 "FIRST_" col 1 char (6),
 4 if (5 = 7), field_2 char (5),
 4 "=THIRD" col 12,
```

yields the result FIRST_␣␣␣␣␣-THIRD which is independent of the length, content, and COLUMN or CHARACTER options of field_2.

9. PICTURE phrase example. Assume that the variable some_data contains the value 123.456 and that it is printed with several different pictures. For example:

```
4 some_data col 21 picture "+999v.999",
```

yields +123.456 in the output line, with the plus sign appearing in column 21.

Picture	Yields
s(3)9v.(3)9	+123.456
(3)9v.(3)9cr	123.456␣␣
(3)9v.(3)9db	123.456␣␣
(5)zv.99	␣␣123.45
$$$$v.99	$123.45
$$$,$$$v.99	␣␣␣$123.45

Report_Field_Def

10. LEFT, CENTER, and RIGHT keywords illustration.

```
4 "pig" char (7) left, yields pigØØØØ
4 "pig" char (7) center, yields ØØpigØØ
4 "pig" char (7) right, yields ØØØØpig
```

Syntax Rule:

The LET keyword cannot be used more than once in defining a report field. Multiple assignment statements may exist within parentheses.

General Rules:

1. The end result of evaluating Char_Expr-4 is a character string. The length of the string depends on the details of the Char_Expr evaluation. The resulting string is a value that is to be placed in the output line.

2. If a LET clause is present, the program must specify one or more assignment statements within the LET clause. The statements in a LET clause are executed as part of executing a PRINT statement in the Execute_Phase group.

3. If a LET clause is present, the items on the lefthand side of the LET assignment statements may be referenced in Char_Expr-4. The LET assignment statements are executed before evaluating Char_Expr-4. (See Examples 3 and 4 in this group discussion.)

4. If the COLUMN integer-16 phrase is used, the value of integer-16 defines the column position to use for the start of the field unless the ALIGN keyword is also used. (See "General Rule 9" if ALIGN is used.)

5. A report field's length is automatically increased by one character in certain situations. Usually, this appears to the user as though a space character is appended to the right-hand end of the report field's value. (See "General Rule 9" for a discussion related to when the character to the right of the report field might be something other than a space.) The report field length is extended by one character is to provide one space between adjacent columns of data without the writer of the source program having to take specific action. The general rules as to when the length extension occurs are stated below, followed by several examples in Table 5-1.

   a. A field is lengthened if its definition does not include the CHARACTER keyword, unless the field value is a literal character string. (A report field defined as a literal character string and without the CHARACTER keyword occupies only the quantity of column positions that the literal occupies.)

b. A field is lengthened if its value is the result of an arithmetic operation and the field definition includes the CHARACTER keyword but does not include the LEFT, CENTER, or RIGHT keywords.

c. A field is lengthened if its definition includes the PICTURE keyword.

d. A field is not lengthened if its value is a non-literal character string and the field definition includes the CHARACTER keyword. The character string may be obtained from the value of an identifier that is declared with either the CHARACTER or the BOOLEAN keyword. The value may also be obtained by concatenating two or more items. Those items may be anything, including decimal literals and decimal identifiers.

The abbreviations used in Table 5-1 are:

dec_id
: Decimal identifier, a user-selected name associated with a local variable or an input field. Its declaration includes the DECIMAL keyword.

char_id
: Character identifier, a user-selected name associated with a parameter, a local variable, or an input field. Its declaration includes the CHARACTER keyword.

num_char_id
: like char_id, but the value is a number.

bool_id
: Boolean identifier, a user-selected name associated with a parameter or a local variable. Its declaration includes the BOOLEAN keyword.

dec_lit
: A decimal, literal value (e.g., 1234).

char_lit
: A character, literal value (e.g., "TOTALS:").

bool_lit
: A boolean, literal value (e.g., true).

char(n)
: The CHARACTER keyword is used in the declaration of the report field and "n" stands for the value that is specified for integer-17.

ASMD
: One of the add, subtract, multiply, or divide arithmetic operations.

||
: The string CONCATENATE operation.

If the LEFT, CENTER, or RIGHT keyword appears in the report field definition, the field is not lengthened.

For the purpose of explaining the examples in Table 5-1, assume that the next few lines appear in the MRPG source program.

Report_Field_Def

```
dcl ddd decimal; dcl ccc character(3); dcl bbb boolean;
ddd := 23; ccc := "cow"; num_ccc := "35"; bbb := true;
```

The meaning of the third and fourth columns of the table is as follows. Consider the first line of the table as an example. In the third column, the report field's definition is assumed to be:

```
4 dec_id,
```

while in the fourth column, the field's definition is assumed to be:

```
4 dec_id char(2),
```

Table 5-1. Automatic Lengthening of a Report Field

Field Description	Example	Is Report Field Lengthened?	
		Field Is Defined As Shown	Field Is Defined With Char(n)
dec_id	ddd	Yes	Yes
dec_lit	56.7	Yes	No
dec_id || dec_id	ddd || ddd	Yes	No
dec_lit || dec_lit	98 || 76	Yes	No
dec_lit ASMD dec_lit	98 + 76	Yes	Yes
dec_id ASMD dec_lit	ddd - 76	Yes	Yes
dec_id ASMD num_char_lit	ddd * num_ccc	Yes	Yes
dec_id ASMD bool_lit	ddd + (true)	Yes	Yes
dec_id picture "picture_string"	ddd pic "99"	Yes	--
char_id	ccc	Yes	No
char_lit	"boy"	No	No
char_lit || char_lit	"cow" || "boy"	Yes	No
num_char_id	num_ccc	Yes	No
num_char_id ASMD num_char_id	num_ccc + num_ccc	Yes	Yes
num_char_lit ASMD num_char_lit	("456") * ("789")	Yes	Yes
bool_id	bbb	Yes	No
bool_lit	(true)	Yes	No
bool_id || bool_lit	bbb || (false)	Yes	No
bool_id ASMD bool_lit	bbb + (false)	Yes	Yes

Report_Field_Def

6. When a numeric literal whose absolute value is less than one is supplied as part of a report field definition, a zero digit is supplied immediately preceding the decimal point. Thus, -.56 is printed as -0.56 while 4¦¦.32 becomes 40.32 with the 0 coming from converting the .32 into a character string so that the concatenation may be done.

7. The BSP option is provided to increase the efficiency of generating and executing an MRPG program. The BSP option need not be used when Char_Expr-4 is a literal, even though the literal contains backspace characters. This is the case on lines 36 and 37 of the example in Section 2. Usually, input information or information created within the MRPG-OS does not contain any backspace characters. Therefore, the usual generation-time and execution-time procedure is to ignore the possibility of backspace characters being present. Use of the BSP option results in additional coding that carries out the complicated steps of properly handling a string that contains backspace characters. If the field is defined using anything other than quoted strings and the values being referenced may ever contain any backspace characters, then the BSP option must be specified.

8. In the ALIGN phrase, string-8 is usually a single ASCII character. That character should exist in the data strings that are to be processed to yield the field being defined (see "General Rule 8"). If the single ASCII character that is wanted for string-8 is the quote character, then four, not three, quote characters must be supplied following the ALIGN keyword. When several output lines that contain a field whose specification includes the ALIGN keyword are produced, the characters in those fields are positioned left or right as needed so that the first occurrence of the string-8 character in those several fields falls on the same column position. When the COLUMN and ALIGN keywords are both specified for the same field, the value of integer-16 is the column number that the string-8 character falls on.

9. This rule regarding ALIGN is independent of the use or omission of the COLUMN phrase. Further, this rule is applicable to all fields whether or not their definition included the ALIGN keyword, but is more likely to come into play for ALIGNed fields because of the left or right sliding that can occur with ALIGNed fields.

   The column positions associated with a field may overlap some or all of the column positions associated with one or more other fields. If so, the earlier information is overlaid, thereby destroying the earlier information.

   No error or warning message is produced at program generation, compilation, or execution time. Thus, the last field to place information into a particular column position takes precedence. The time sequence in which fields are placed into an output line corresponds with the physical sequence in which the definitions of the fields occur in the source program (see Example 7).

10. Consider a field whose definition includes an ALIGN phrase. If the string-8 character does not occur in the character string that is to be placed in an output line, the character string is aligned as though the string-8 character immediately followed the character string.

11. Use of the PICTURE "string-3" option provides extensive flexibility for converting numerical data into a modified representation in a report field. Whatever is supplied as string-3 is checked for validity using the PL/I compiler's standard picture verification subroutines. Assuming that no errors are detected, string-3 is passed to the PL/I compiler as part of the generated PL/I source program. Thus, the full PL/I picture editing capability is available to an MRPG program. (This is not the full PL/I picture capability. The PL/I picture encoding capability, used in PL/I programs to convert numerical data into a

Report_Field_Def

character representation that is placed in storage for further use, is not accessible to the MRPG language.) Since the PL/I picture editing capability is extensive and may change with time, describing it here is beyond the scope of this manual. The reader should refer to the Multics PL/I manuals for complete details. This is a PL/I characteristic.

12. When the CHARACTER option is chosen, integer-17 specifies the quantity of column positions available to the field.

13. The LEFT, CENTER, and RIGHT options specify that, when the output character string is shorter than the field width specified by integer-17, the output string is to be left-justified, centered, or right-justified, respectively, in the output field area. Spaces fill any unused column positions. If the character string to be placed in the report field is interpreted by the MRPG-OS as being a numerical value and the LEFT, CENTER, or RIGHT keyword is not present, the default is RIGHT. If the character string is not a numerical value, the default is LEFT.

The_MRPG_Program

Group Name: The_MRPG_Program

This page shows the overall structure of a source program. The remainder of this section expands the general format shown below into a complete, detailed specification of the language.

General Format:

```
[ Declare_Parameters ]
Declare_Input_File
[ Declare_Variable ] ...
{ Define_Report } ...
{ Execute_Phase } ...
END ;
```

Examples:

1. See Section 2 for a complete example, including input data, source program, and the report.
2. See Appendix B for additional examples.

Syntax Rules:

1. Note that the Declare_Input_File, Define_Report, and Execute_Phase groups are required. The other two groups are optional. An "end;" is also required.
2. The three ellipses mean that the Declare_Variable, Define_Report, and Execute_Phase groups may be repeated (i.e., their general formats may be used over and over).
3. Whichever groups are used must be used in the order shown above.

General Rules:

1. The Declare_Parameters group is used whenever the MRPG-OS needs to be able to accept parameters.
2. The Declare_Input_File group is used to describe the structure of the input file, to identify those fields to be used by the MRPG-OS, and to assign attributes to the fields.

The_MRPG_Program

3. The Declare_Variable group is used whenever some information not explicitly contained in the input file must be constructed and saved for use at a later point in executing the MRPG-OS. However, the use of control breaks does not require the inclusion of the Declare_Variable group.

4. The Define_Report group is used to describe the layout of the reports, control breaks, and where the MRPG-OS is to send the reports.

5. The Execute_Phase group contains the statements that trigger the actual printing of the detail lines of the report(s). The printing of headings and footings occurs automatically as detail lines are created. Calculations that assign values to local variables can reside in the Execute_Phase group of a program. Sorting may also be specified in this group.

SECTION 6

DATA TYPES

This section contains general information about data types and conversion between data types.

As in earlier sections, other forms of a keyword also apply when one of the forms is used in this section. Thus, CHARACTER means both CHARACTER and CHAR, AND means both AND and &, and LE (less than or equal to) means both LE and <=.

TYPES OF DATA

The set of keywords that identify data types and are pertinent to data conversion are:

DECIMAL
CHARACTER
BOOLEAN
PICTURE

After a brief discussion of each of the above data types, the rules governing the combining of and conversion between data of different types are presented.

Usually, the implementation details of which PL/I attributes apply to the variables in an MRPG source program is of no concern to the programmer. However, if it becomes necessary to know the PL/I attributes, they can be obtained from an examination of the PL/I source produced by the MRPG. Or, if the PL/I control arguments that were received by the MRPG and passed on to the PL/I compiler caused a PL/I listing to be produced, the PL/I attributes may be obtained from the listing. It is easier to obtain the attributes from the .list segment than from the .pl1 segment, but there is ordinarily no need to have the PL/I compiler perform the extra work required to produce a listing.

In this section, the term "variable" is used in its general sense of referring to a data item whose value may be different at different times. That is, "variable" is not restricted to local variables.

Integers, numbers, and variables declared with the keyword DECIMAL can all be thought of as being kept and treated as decimal data when these items are used in an arithmetic sense. The limits on the sizes and resolution of integers, numbers, and decimal variables is determined by the MRPG implementation.

INTEGERS

Integers are handled as fixed bin(35) data, which implies that the range of integers is:

```
     -2**35    = -34,359,738,368
to
      2**35-1 =  34,359,738,367
```

where ** denotes exponentiation.

However, the limits on the sizes of integers are far smaller than indicated here. These limits vary with the use of the integers and are fully discussed in Section 5.

NUMBERS AND DECIMAL VARIABLES

Numbers and decimal variables are handled as float decimal(20) data. This is true even when the number or decimal variable has an integral value. This float decimal(20) form is used even if the number or decimal variable does not contain an explicit decimal point, that is, the value being assigned is equal to an integer. With this representation, values up to 20 digits in length may be used. Thus, the following are all valid assignment statements in an MRPG source program:

```
amount :=  .00000000000000000001;
amount := +.12345678901234567899;
amount :=   5.98;
amount := -1234567890.1234567899;
amount :=   99999999999999999999.;
```

The limit of 20 digits is just that, 20 digits, not 20 characters. That is, a sign character and a decimal point may be present in addition to the 20 digits. Thus, up to 22 characters can be used to specify a decimal value and all 20 digits of significance are retained.

It is true that the float decimal(20) form can accommodate much larger and much smaller values than the values shown above. A 30-digit integral value could be supplied, but only the most significant 20 digits are retained. Rounding of the 20th digit may occur. If this value is placed in a report field, the last 10 digits (least significant) are zeros.

Character Data

The keyword CHARACTER and the "string-n" construct appear throughout Section 5. The associated data are strings of ASCII characters. The upper limit on the length of a character string is the PL/I limit of 256 characters. However, the rules in Section 5 restrict most character strings to much shorter lengths.

As in PL/I, the appearance of the VARYING keyword with the CHARACTER keyword indicates that the associated string's length may change. The current length is carried along with the string. The decision whether to supply or omit the VARYING keyword depends on how the variable is used. Consider the printing of a line that has the following layout:

```
Inventory cost of <part_name> is <part_inventory_cost>.
```

Assume that part_name is declared char(11) (i.e., without the VARYING keyword) and let xxx represent the value of the part_inventory_cost variable. The printed line for the values shown in part_name are:

part_name value	The Printed Line
screwdriver	Inventory cost of screwdriver is xxx.
hammer	Inventory cost of hammer is xxx.
saw	Inventory cost of saw is xxx.

If, however, the VARYING keyword is included in the declaration of part_name, then the printed lines are:

part_name value	The Printed Line
screwdriver	Inventory cost of screwdriver is xxx.
hammer	Inventory cost of hammer is xxx.
saw	Inventory cost of saw is xxx.

Boolean Data

The MRPG programmer may think of the value of a boolean variable as being either true or false. The actual implementation is done with a bit string that is one bit long.

Picture Data

The PICTURE keyword is used only with report fields. The result of performing the transformation specified by the picture string is a character string. The length of the result is determined by the quantity of columns represented by the picture, not necessarily the quantity of picture indicators in the string that specifies the picture. (Refer to the discussion of pictures in the PL/I manuals for details.)

The following conversion discussions specify what happens when various types of conversions are called for by an MRPG source program, either implicitly or explicitly. If the final usage of a value is in a report field, the value must be in the form of a character string, since only ASCII characters are placed into report fields. Therefore, if a value is not a character type, but is to be placed into a report field, conversion to a character string occurs.

A common conversion situation arises in an assignment statement such as:

```
target := source;
```

but other situations exist and are discussed in the next few paragraphs.

A report field definition of the form:

```
4 char_item * bool_item  picture "<a picture string>",
```

requires three conversions. Suppose char_item is a character variable and bool_item is a boolean variable. The values of char_item and bool_item must be converted to decimal so that the multiplication can be performed. The sum must then be converted into a character string according to the picture indicators in the picture string.

It may be possible to determine at generation time or at compilation time that the conversion cannot succeed. If so, an error message is produced. Usually, though, a conversion failure does not occur until execution time. Then, the conversion condition is signalled. Refer to the PL/I manuals for discussions of signalling and conditions. Usually, a conversion condition at execution time results in control being returned to command level. The probe or debug tools may be used to investigate.

There is no discussion of converting to or from an integer type of data because there is no provision in the language for requesting such a conversion.

The conversion of a boolean value through a picture into a report field is not supported, and therefore, is not discussed.

Decimal to Character

There are four possible types of targets into which a decimal value may be converted:

- Local variable declared with CHARACTER keyword
- Input field declared with CHARACTER keyword. The input field must be "held" by a HOLD statement. (See "Execute_Phase" group in Section 5.)
- A report field that is neither a PICTURE nor an EDIT field
- A PICTURE report field

PICTURE conversions are covered in the PL/I manuals. The first three target types are covered here.

The rules that specify the fundamental conversion are the same in all three cases. There is some variation in what happens should certain abnormal or error conditions arise. The rules pertinent to justification within the target area and any associated padding with spaces vary considerably, depending on the target type. Several examples are given following the rules.

1. For purposes of explanation, assume that the decimal value is first converted into a temporary character string. Next, justification and space padding may occur as the temporary string is placed into the target area.

2. If the decimal value is zero, the conversion is complete. The temporary string has zero length. Advance to rule 7.

3. If the decimal value is negative, the first character of the temporary string is a minus sign. If the decimal value is positive, no character is placed in the temporary string at this point.

4. If the decimal value is equal to or larger than 1, the decimal digits that comprise the integral part of the decimal value are concatenated onto the temporary string.

5. If the decimal value is less than 1, a zero is concatenated onto the temporary string.

6. If the decimal value has a fractional part, a decimal point and those fractional decimal digits are concatenated onto the temporary string. There are no zeros in the temporary string after the least significant fractional non-zero digit.

7. The fundamental conversion is complete. Now justification and/or space padding may occur. Advance to the rule indicated below:

Rule No.	Target Type
8	Local variable
8	Input field
12	Report field

8. The length of the temporary string is compared to the maximum allowable length of the local variable or input field to determine whether or not the temporary string can fit into the target string's area.

9. If the temporary string can fit, then:

   a. If the temporary string is shorter, and if the target string's declaration included neither the VARYING nor the SPECIAL keyword, then sufficient spaces are concatenated onto the temporary string to make its length equal to the length of the target string. Then the temporary string is placed into the target string.

   b. If the temporary string is shorter, and if the target string's declaration included either the VARYING or the SPECIAL keyword, then the temporary string is placed in the target string. The length of the target string is set to the length of the temporary string.

10. If the temporary string is too long to fit into the target string, the temporary string is truncated to the length of the target string. No warning or error message is produced. The truncated temporary string is stored in the target area.

11. The total conversion for local variables and input fields is now complete.

12. The final actions in converting a decimal value into characters in a report line involve three major steps. First, the temporary string that existed in step 7 is converted into a "report string", which is the string of characters that are placed into the report line. Second, the position of the report string in the report line is determined. And third, the report string is actually stored into the report line. Keep in mind that if any overstriking exists in the report string, the quantity of characters in the report string are larger than the quantity of columns that are to be occupied in the report line. Let "report_columns" stand for the quantity of columns that are to be occupied in the report line.

13. Think of the report line as being filled with spaces before any report fields are stored into the report line. As each field is stored into the line, the previous contents of the column positions stored into are destroyed.

14. If the report field definition included the CHARACTER keyword, then integer-17 specifies the maximum value of report_columns. Assuming this to be the case, the length of the temporary string is compared to report_columns.

    a. If the quantity of columns represented by the temporary string is less than or equal to report_columns, then the report string's content and length are the same as for the temporary string. It is possible for the temporary string to be short enough so that not all of the column positions implied by report_columns are filled.

    b. If the quantity of columns represented by the temporary string is larger than report_columns, every character in the report string is set to the # character. There are integer-17 such characters.

15. If the report field definition omitted the CHARACTER keyword, then the content and length of the report string are set to the content and length of the temporary string.

16. At this point, the content and length of the report string are established. The column positions that the report string is to occupy depend on the presence or omission of several keywords in the report field definition. The complete details are given in the Report_Field_Def group in Section 5 and are not repeated here. In rereading that material, notice that the decimal value to report string conversion described in this section says nothing about the starting column number or limits on the width of the report field into which the report string is to be placed.

17. Sometimes, when a decimal value is converted into a character string and placed into a report field, the report field is lengthened by one character. Usually, this gives the appearance in the printed output of an "extra" space added to the end of the field. This topic is covered in depth in the Report_Field_Def group in Section 5. Within the following table, the report field values shown do <u>not</u> include the effect of this possible field extension.

Table 6-1. Examples of Decimal to Character Conversion

Decimal Value	Fixed Length Local Variable or Input Field			Varying Length Local Variable (varying) or Input Field (special)		
	Description	Result	Length	Description	Result	Length
456.89	char(7)	456.89Ø	7	char(7) var	456.89	6
456.89	char(6)	456.89	6	char(6) var	456.89	6
456.89	char(5)	456.8	5	char(5) var	456.8	5
.5678	char(7)	0.5678Ø	7	char(7) var	0.5678	6
.5678	char(6)	0.5678	6	char(6) var	0.5678	6
.5678	char(5)	0.567	5	char(5) var	0.567	5
0	char(5)	0ØØØØ	5	char(5) var	0	1
-.567	char(7)	-0.567Ø	7	char(7) var	-0.567	6
-.567	char(6)	-0.567	6	char(6) var	-0.567	6
-.567	char(5)	-0.56	5	char(5) var	-0.56	5
-23.56	char(7)	-23.56Ø	7	char(7) var	-23.56	6
-23.56	char(6)	-23.56	6	char(6) var	-23.56	6
-23.56	char(5)	-23.5	5	char(5) var	-23.5	5

Decimal Value	Report Field Without the CHARACTER option		Report Field with the CHARACTER Option, All Starting In Same Column		
	Result	Length	Description	Printed Result	Length
456.89	456.89	6	char(7)	456.89Ø	7
456.89	456.89	6	char(6)	456.89	6
456.89	456.89	6	char(5)	#####	5
.5678	0.5678	6	char(7)	0.5678Ø	7
.5678	0.5678	6	char(6)	0.5678	6
.5678	0.5678	6	char(5)	#####	5
0	0	1	char(5)	0ØØØØ	5
-.567	-0.567	6	char(7)	-0.567Ø	7
-.567	-0.567	6	char(6)	-0.567	6
-.567	-0.567	6	char(5)	#####	5
-23.56	-23.56	6	char(7)	-23.56Ø	7
-23.56	-23.56	6	char(6)	-23.56	6
-23.56	-23.56	6	char(5)	#####	5

Character to Decimal

This discussion applies to those situations wherein the source is a character string and the target is a decimal value.

The character string can be created in many ways, for example:

- A parameter
- Local variable declared with CHARACTER keyword
- Input field declared with CHARACTER keyword
- Conversion of the result of arithmetic operations to a character string
- Output of a TRANSFORM table lookup
- Result of invoking certain builtin functions (e.g., %substr (%yyddd,3,3), which is the number of the day in the year)

There are several possibilities for the target, for example:

- Local variable declared with DECIMAL keyword
- Input field declared with DECIMAL keyword. The input field must be "held" by a HOLD statement. (See "Execute_Phase" group in Section 5.)
- Input value to a SET or TRANSFORM table lookup
- An arithmetic operand in an expression

However it is created, the character string has a value and a length. It may contain leading or trailing spaces, which are ignored. (This is an PL/I characteristic.) If the character string to be converted contains letters, an error occurs. However it is going to be used, the decimal value is a float decimal(20) value. The following paragraphs specify the conversion rules, including the determination of what constitutes a valid character string.

If any of the following rules are violated, the conversion fails. The error may be detected by the MRPG or the PL/I compiler, in which case an error message is displayed. If the error is detected during the execution of the MRPG-OS, the conversion condition is signalled.

1. The only valid characters are the 10 decimal digits, the space, the period (decimal point), the plus, and the minus.
2. No more than one period may be present.
3. No more than one arithmetic sign character (plus, minus) may be present. If one is present, it must immediately precede the leftmost digit.
4. The only non-digit character permitted between the leftmost digit and the rightmost digit is the period.
5. The only non-digit character permitted between an arithmetic sign and the leftmost digit is a period.
6. If a minus sign is present, it must be followed by at least one digit.

The conversion is made in accordance with the following rules. Some examples follow these rules.

1. A series of consecutive spaces at the beginning of the string is ignored.

2. A series of consecutive spaces at the end of the string is ignored.

3. If a period is present and there is at least one non-zero digit to the left of the period, a series of zeros preceding the leftmost non-zero digit is ignored.

4. If a period is present and there are no non-zero digits to the left of the period, a series of consecutive zeros to the left of the period is ignored.

5. If a period is present and there is at least one non-zero digit to the right of the period, a series of consecutive zeros to the right of the rightmost non-zero digit is ignored.

6. If a period is present and there are no non-zero digits to the right of the period, a series of consecutive zeros to the right of the period is ignored.

7. If none of the digits 1 through 9 are present, the target value is zero. There is no distinction between a positive zero and a negative zero. The positive representation is used.

8. If no period is present, the decimal point is considered to immediately follow the rightmost digit.

9. If a minus character is present, the target value is negative. If no minus character is present, the target value is positive.

10. If the quantity of significant (non-ignored) digits is greater than 20, only the most significant 20 are retained. Rounding occurs.

Table 6-2. Examples of Character String to Decimal Value Conversion

Source String	Source Length	Target Value (as it is printed in a report field)
ßßß123.56ßßß	12	123.56
0001.2345000	12	1.2345
000.12300	9	0.123
1234.0000	9	1234
1234.	5	1234
1234	4	1234
-000.0000	9	0
-0001.2	7	-1.2
ßßß	3	0
ßß-0ßß	6	0
-1234567890123456.7890987	25	-1234567890123456.7891
78901234567890123456789 01	25	7890123456789012345700000
ßß-ßß	5	error
5x6	3	error
5.6.	4	error
-5+6	4	error
000+5.6	7	error
5ß6	3	error
-ß5.6	5	error
12,345.67	9	error

Decimal to Boolean

If the decimal value is zero, the boolean value is set to false. Otherwise, the boolean value is set to true.

Boolean to Decimal

If the boolean value is true, the decimal value is set to 1. If the boolean value is false, the decimal value is set to 0.

Character to Boolean

The boolean value is set to false unless all of the following constraints are satisfied, in which case the boolean value is set to true.

1. The four-letter English word "true" appears in the character string. The letters of the word "true" may be in any combination of uppercase and lowercase characters. For example, "TRUE", "TruE", and "tRUe" are all valid representations of the word "true".

2. The only other character that is in the string is the space character. It is valid for spaces to precede, follow or both precede and follow the "true" word.

3. The four letters of the word "true" must be contiguous.

Boolean to Character

If the boolean value is true, then the character value is set to the four characters "true" and, if the length is variable, the current length is set to four. If the boolean value is false, then the character value is set to the five characters "false" and, if the length is variable, the current length is set to five. In both the true and false cases, if the target string is non-variable and is longer than the four or five characters needed to hold the "true" or "false" characters, trailing spaces are supplied to fill out the target string.

If the "true" or the "false" string is too long to fit into the target, the "true" or "false" string is truncated.

Decimal to Picture

The rules are described in the PL/I manuals.

Character to Picture

If the character string has a numerical value, the source value is converted to a decimal value which is then converted into the picture value according to the rules described in the PL/I manuals.

If the value of the character string is not a number, an error occurs.

SECTION 7

EXPRESSIONS

The main purpose of this section is to integrate the five groups of Section 5 that specify the MRPG's expression capability. These five groups are:

- Full_Expr
- Relationship_Test
- Char_Expr
- Char_Ref
- Arith_Expr

The next portion of this section identifies the types of expressions. Then, the interactions between the above five groups is examined. Following that, the individual operators that may be used to combine expressions are discussed, along with their precedence rules. Finally, some rules governing the allowable combinations of operators and expressions are described.

TYPES OF EXPRESSIONS

An expression is one of the following:

literal
variable reference
builtin function reference
operator expression

Each of the above consists of a few or several items. The grouping of the items is listed below. The specifications for the items are in Section 5.

A literal may be an integer, a number, or a character string.

A variable reference may be a parameter name, an input field name, or a local variable.

The set of builtin function reference possibilities is listed below. Many of these have arguments, which are not shown. Strictly speaking, the TRUE and FALSE keywords are not functions, because the keyword itself is the value. However, it is convenient to think of TRUE and FALSE as builtin functions. The column headings indicate which type of data value is returned.

Arithmetic	Boolean	Character
%PAGENUMBER	IN	TRANSFORM
	TRUE	%MMDDYY
	FALSE	%YYDDD
	%LEVEL	%MONTH
		%DAY
		%HHMMSS
		%SUBSTR
		%REPEAT

An operator expression performs some operation on its operand(s) and delivers the result as the value of the operator expression. An operator expression has one of the following forms:

- prefix_operator expression
- expression infix_operator expression

When used for arithmetic operations, prefix operators are also known as unary arithmetic operators, and infix operators are also known as binary arithmetic operators.

A few examples of operator expressions are:

Prefix Type	Infix Type
+5.3	count_total + count
-discount_rate	output_file_name || ".report"
^color_match	weight <= 500
-(gross_salary * charity_deduction)	gross_salary * charity_deduction

INTERACTIONS BETWEEN EXPRESSION-RELATED GROUPS

The Full_Expr group's definition includes two other groups (Char_Expr and Relationship_Test). Examination of all five groups shows that a circular definition path exists. Figure 7-1 summarizes how the five groups are related.

Each group definition includes options. In most groups, one or more of the options includes a reference to another group. For example, the FILE Char_Expr-2 in the Declare_Input_File group signifies that the keyword FILE must be followed by an expression that satisfies the specifications of the Char_Expr group. Choosing such an option means that some option is to be selected from the indicated subsidiary group.

The majority of groups also includes options that do not reference other groups, e.g., the STREAM option in the Declare_Input_File group. This type of option is called a terminating option, and when selected, a chain such as the following is terminated:

```
select an option that involves another group
and in it select an option that involves another group
and in it select an option that involves another group
  .
  .
  .
```

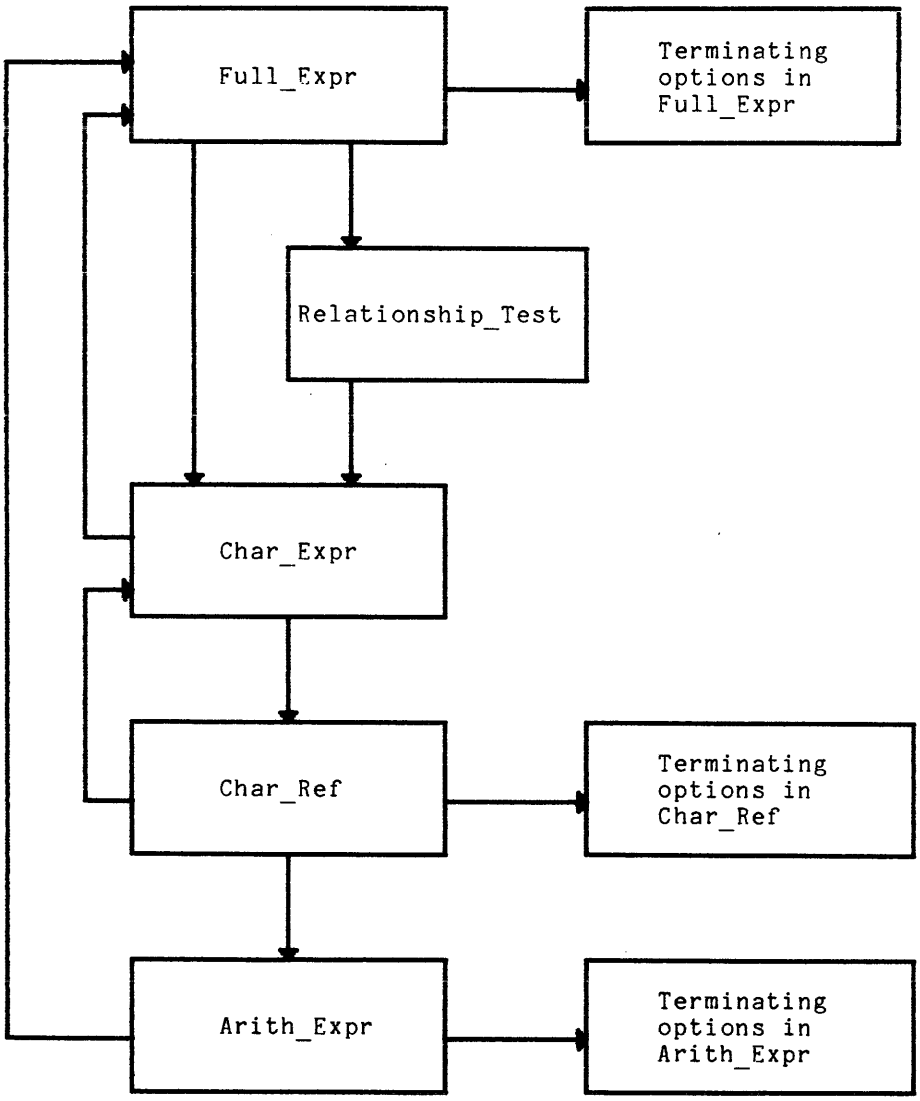


Figure 7-1. Expression-Related Groups Interdependence

The Relationship_Test group should really be thought of as part of the Full_Expr group. The Relationship_Test group appears in a separate block in Figure 7-1 and as a separate discussion in Section 5 as a convenience in showing the general format diagrams and to reduce the quantity of pages required to specify one group.

The Full_Expr, Char_Expr, Char_Ref, and Arith_Expr groups must be kept as separate groups. In addition to the group interconnections shown in Figure 7-1, each of these groups appears in the definition of one or more of the other groups that make up the full MRPG language specification.

Because of the circularity depicted in Figure 7-1, expressions may contain other expressions. Such nesting may be made to any depth. Parentheses may be required to write a complicated expression. The Arith_Expr group's definition formally provides the parentheses.

OPERATORS

In the following text, the operators are grouped in classifications. The reader is referred to the appropriate Group in Section 5 for the meaning of each operator, the precedence among the operators is specified, and the result of using parentheses is specified.

Operator Classification

Operators are used with expressions either to obtain new data values or to make tests. Table 7-1 classifies all the operators.

The term data value refers to an arithmetic, character, or boolean value. The value may or may not change while an MRPG-OS is being executed. The value may be in the input, the output, or be calculated by the program. Calculated values may not be visible to the program because they are temporary values used for further calculations or testing.

The Quantity column in Table 7-1 specifies how many operators appear on each line of the table. Note that some rows of the table contain more than one line. The Operators column contains the ASCII character or characters that identify the operator. If the value in the Quantity column is greater than 1, spaces separate the operators. The Position column specifies whether the operators are to be thought of as prefix operators for one expression or infix operators between two expressions. The Usage column states how the operators are used.

Table 7-1. Operator Classification

Quantity	Operators	Position	Usage
4	+ - NOT ^	prefix	data value
4	+ - * /	infix	data value
2	CONCATENATE ||	infix	data value
4	OR | AND &	infix	data value
6 6	LT LE EQ GE GT NE < <= = >= > ^=	infix	test -- value comparison
1 1 1 1 1 1 1 1 1 1	BEGINS CONTAINS ENDS NOT BEGIN ^ BEGIN NOT CONTAIN ^ CONTAIN NOT END ^ END	infix	test -- string matching
1 1 1 1 1 1 1 1 1 1	BEGINS WORD CONTAINS WORD ENDS WORD NOT BEGIN WORD ^ BEGIN WORD NOT CONTAIN WORD ^ CONTAIN WORD NOT END WORD ^ END WORD	infix	test -- word matching
1 1 1	IN NOT IN ^ IN	infix	test -- set membership

Operator Meanings

The meaning of all operators is specified in Section 5. Each operator has a normal data type. Section 5 specifies the normal data type for each operator and describes what happens when other data types are encountered. The groups in which the operators are discussed in Section 5 are listed below.

Operator	Group
+ - * /	Arith_Expr
CONCATENATE ||	Char_Expr
OR | AND & NOT ^	Full_Expr
All other	Relationship_Test

Operator Precedence

Two operators are on the same expression level if they appear in the same expression and only matched pairs of parentheses appear between the two operators. When there is more than one operator at the same expression level, the operator with highest priority is evaluated first. The priority ranking is shown in Table 7-2. If there is more than one operator with the same priority, they are evaluated either from left to right or right to left within the expression, as specified in Table 7-2.

Table 7-2. Operator Precedence

Priority	Operators	Order Within Priority
highest	NOT ^ prefix + prefix -	right to left
next	* /	left to right
next	infix + infix -	
next	CONCATENATE ||	
next	LT < LE <= EQ = GE >= GT > NE ^= BEGINS CONTAINS ENDS NOT BEGIN NOT CONTAIN NOT END BEGINS WORD CONTAINS WORD ENDS WORD NOT BEGIN WORD NOT CONTAIN WORD NOT END WORD	
next	AND &	
lowest	OR |	

Parentheses

Any expression may be enclosed in parentheses. These parentheses are in addition to any parentheses required by the general formats in Section 5. Providing extra parentheses can put operators at different expression levels, thereby changing the sequence in which the operators are evaluated.

COMBINING DATA VALUE EXPRESSIONS

Tables 7-3 and 7-4 summarize the ways in which expressions, operators, and parentheses may be combined to form a more complicated expression. Within these tables, the term "expr" denotes any expression that is to be combined with some other expression to yield a new data value. That "expr" may contain operators and parentheses. The character string +- refers to the prefix arithmetic operators. The string +-*/ refers to the infix arithmetic operators. The NOT refers to the negation operator of the Full_Expr group. In the Location column, "first" means that the element being considered is the first element in the combined expression that is being written. The term "inside" means that the element is neither the first nor the last one. The term "last" means that the element is the last element in the new, combined expression.

Table 7-3. Combining Arithmetic Expressions

Valid Preceding Elements	Elements Under Consideration		Valid Following Elements
	Location	The Elements	
none	first	expr	+-*/
		+-	expr (
		(	+- expr (
+-*/ +- (	inside	expr	+-*/)
expr)	inside	+-*/	+- expr (
+-*/ (	inside	+-	expr (
+-*/ +- (	inside	(	+- expr (
expr)	inside	)	+-*/)
+-*/ +-	last	expr	none
expr)	last	)	

Table 7-4. Combining Logical Expressions

Valid Preceding Elements	Elements Under Consideration		Valid Following Elements
	Location	The Elements	
none	first	expr	OR AND
		NOT	expr (
		(	expr (
OR AND NOT (	inside	expr	OR AND)
expr)	inside	OR AND	expr NOT (
OR AND (	inside	NOT	expr (
OR AND NOT (	inside	(	expr NOT (
expr)	inside	)	OR AND (
OR AND NOT	last	expr	none
expr)	last	)	

SECTION 8

PROGRAM PREPARATION, GENERATION, AND EXECUTION

PROGRAM PREPARATION

The program preparation portion of this section offers some thoughts on functional requirements of the MRPG program, presents some suggestions on the program's design, and briefly mentions how to enter the source program and save it for use by the MRPG.

Initial Decisions

Several decisions should be made before beginning the detailed design of the program. The following topics may have significant impact on the general approach used.

INVOCATION METHOD

If the object program is invoked from Multics command level, or via an attachment to the report_ I/O module, or both ways at different times, then portions of the program as well as documentation for users of the program are affected.

When the MRPG-OS is invoked from command level, either the FILE or the ATTACH option must be selected in the Declare_Input_File group. Users who invoke the MRPG-OS must know the pathname of the MRPG-OS.

If the MRPG-OS is invoked via an attachment to the report_ I/O module, the FILE and ATTACH options may be omitted. If one is present, it is ignored. (One of them may be present if the MRPG-OS is to be invoked as a command at some other time.) The SPECIAL keyword must be included in the definition of every input field. Users of the MRPG-OS may or may not need to know where the MRPG-OS is located in the virtual memory. The need to know the MRPG-OS pathname is dependent on the program that invokes the MRPG-OS.

LOCATION OF INPUT DATA

If the input data is in a segment in the virtual memory, then the pathname of that segment may need to be included in the program. However, it is possible to obtain the input from different segments at different times by using the value of one or more parameters in constructing the segment's pathname.

If the I/O attachment method is used, it is possible that the input data may never exist as a segment in the virtual memory.

LOCATION OF OUTPUT REPORTS

The program specifies where each report is sent. The output may be written to a segment using the FILE and SWITCH options in the Report_Control group. If no FILE or SWITCH options are specified, or if none of those specified are selected by an associated IF test, or if the selected SWITCH option specified the user_output switch, the report is sent to the user output I/O switch. In this case, the lines of the report usually appear on the user's terminal as the lines are produced, with no copy of the report available for reprinting at a later time. Actions external to the MRPG-OS can divert user_output traffic to a segment, (e.g., the file_output command can do this).

OUTPUT PRINTING METHOD

If a hardcopy version of a report is wanted, several choices exist. Simple printing on a terminal may be adequate, but if the lines in the report are longer than the width of the available terminal, it may be necessary to use a line printer of adequate width. The line printer may be located at the central computer site or be remote with the data transmitted over a data communication line. High quality printing may be obtained by using a special ribbon in a line printer, a special ribbon in a terminal, or via COM (computer output on microfilm). The particular printing method used may affect the page layout information supplied in the Report_Control group and possibly the PAUSE or 0 options in the Line group. Section 9 provides additional details on printing reports.

PARAMETERS

If the program is to utilize parameters, then a decision as to their acceptable values, any constraints on their input sequence, and the use of keywords must be made. These decisions are implemented with the source program text in the Declare_Parameters group and in whichever groups use the values that are supplied at execution time for the parameters.

VALIDITY CHECKING

It is possible to provide extensive validity checking on parameters and input fields. The thoroughness desired must be decided on, as well as the details of each individual check. In general, making a validity check is inexpensive.

Detailed Program Design

The next few paragraphs offer some thoughts that should be considered at the time that the program structure is being defined.

MULTIPLE USE OF INPUT FILE

If a file is used as input to several MRPG programs, then it is recommended that a complete description of the file be worked out and saved in the virtual memory. When a new program is written that uses this input file, the file's description can be copied from the saved file into the proper place in the source program.

INPUT FILE STRUCTURE

If a complete file description exists for the input file and is stored in the system, that file description can be copied into the source program. Otherwise, a new file description must be prepared. Unused fields need not be described. They can be skipped over with the FILL option of the Declare_Input_File group or the POSITION option of the Input_Field_Def group. Or, an artificial padding field whose size equals the sum of the sizes of the contiguous fields to be skipped may be specified. Characters at the end of records in a stream file are skipped over if these character positions are omitted from the file's declaration. (See Figure 2-1.)

OUTPUT REPORT LAYOUT

When specifying the positioning of data in an output line, it may be helpful to think of a line as initially consisting of all spaces. Data placed into the line merely overlays what was already there. If some data is specified as occupying column positions also occupied by some other data, the final data is the data that was specified last. That is, a line is built up in the sequence in which the specifications of the data items occur in the source program. After all fields have been placed in the line, trailing spaces are removed.

Typing in the Source Program

The source program is physically entered into the computer system with a text editor. The mechanics of editing and correcting the characters that comprise the source program are completely determined by the particular text editor used.

Saving the Source Program

The source program must be placed in a segment in the virtual memory in order for the MRPG to generate an object program. Therefore, after the source program is built with the aid of a text editor, the source program must be written into a segment.

The segment must be given a two-component name, the second of which is the five-character .mrpg suffix. The first component must start with a letter. Only the 52 letters, the 10 digits, and the underscore characters may be used to form the first component. This is a PL/I characteristic.

The maximum length of the segment's name, including the .mrpg suffix, is 32 characters. This is a Multics characteristic.

GENERATING AN OBJECT PROGRAM

The actions involved in converting an MRPG source program into a standard Multics executable object segment are described below.

Invoking the MRPG

The MRPG is invoked by issuing the mrpg command from Multics command level. The mrpg text follows the command writeup format used in the MPM Commands.

Name: mrpg

The mrpg command invokes the MRPG to translate a segment containing MRPG source statements into a segment containing PL/I source statements. Then the PL/I compiler is automatically invoked to translate the segment containing PL/I source statements into a standard Multics object segment. PL/I control arguments may be supplied with the mrpg command. These PL/I control arguments are passed on by the MRPG to the PL/I compiler. The results are placed in the user's working directory. The mrpg command cannot be called recursively. For information on PL/I, refer to the PL/I manuals.

Usage

```
mrpg path { PL/I control_args }
```

where:

1. path
   is the pathname of an MRPG source segment that is to be translated by the MRPG. If path does not have a suffix of mrpg, then one is assumed. However, the suffix mrpg must be the last component of the name of the source segment.

2. PL/I control_arguments (optional)
   can be chosen from the list of control arguments for the pl1 command in the MPM Commands.

   No checking is done by the MRPG on the supplied PL/I control arguments.

Notes

The PL/I source segment produced by the MRPG is placed in the user's working directory. This segment's name is the same as the name of the segment supplied as input to the MRPG except that the mrpg suffix is replaced with a pl1 suffix.

The PL/I source segment is not deleted by the MRPG.

The object segment produced by the PL/I compiler is placed in the user's working directory. This segment's name is the same as the name of the original source segment with the mrpg suffix omitted.

Execution of the MRPG generation and the PL/I compilation may be interrupted at any time by pressing the Quit/Interrupt/Break switch on the terminal. Typing "start" will cause execution to resume at the interrupted point. The program_interrupt feature is not supported.

Error Diagnostics

The MRPG diagnoses and issues error messages via the error_output I/O switch using the three levels of severity:

1 Warning only. The term *WARN identifies this class of error messages. The generation of PL/I source statements proceeds without ill effect. The assumptions made by the MRPG are reported.

2 Correctable error. The term ERROR identifies this class of error messages. The MRPG makes the best attempt that it can to rectify the situation and continues. The correction made is reported. In many cases, the correction made is the same change that the programmer would make. In any event, generation of PL/I source statements continues so that as many errors as possible can be reported during each pass through the MRPG source.

3 Uncorrectable error. The term FATAL identifies this class of error messages. The MRPG cannot determine what might be a reasonable correction and skips forward in the source statements to a point at which it may be possible to again generate meaningful PL/I source statements. The resulting PL/I source program is not correct.

After the MRPG has completed processing its input, the maximum severity level error detected is tested. If the maximum severity level is 3, control returns to the Multics command processor. If the maximum severity level is none, 1, or 2, the PL/I compiler is invoked.

Error messages may be produced by the PL/I compiler. Their severity levels are discussed in the MPM Commands.

Because the MRPG allows numbers to be used as character strings and allows a character string whose content is a number to be used in an arithmetic value, some conversions between different PL/I data types occurs. Such conversions do not result in any warning messages from the MRPG, but may result in warning messages from the PL/I compiler. These PL/I warnings should be ignored. They can be suppressed by supplying the control argument -severity1 when the MRPG is invoked. That control argument is passed to the PL/I compiler and inhibits all error messages whose severity level is 1. This is a PL/I characteristic.

If PL/I error messages with severity 2 or higher persist after the source program is changed to eliminate all MRPG error messages, contact the local Honeywell representative for assistance.

The MRPG does not support the severity active function.

Listings

The MRPG does not produce a listing, but if desired, a PL/I listing can be obtained. How to do this and how to interpret the PL/I listing are discussed in the MPM Commands and the PL/I manuals.

PROGRAM EXECUTION

The MRPG-OS is invoked from command level by supplying a normal Multics command line:

name_of_the_MRPG-OS {parameters needed by the MRPG-OS}

Each parameter delimited by white space or the end of the command (i.e., the last parameter may be followed by white space, a newline, or a semicolon).

If no parameters are required, the command line consists of just the name of the MRPG-OS.

When the MRPG-OS completes execution, control is returned to the command processor.

As with any Multics command, program execution may be interrupted by pressing the Quit/Interrupt/Break switch on the terminal. Typing "start" causes execution of the MRPG-OS to be resumed at the point at which the interruption took place. The program_interrupt feature is not supported.

If the PL/I -table control argument is supplied with the mrpg command, the probe debugging command may be useful should errors occur during program execution.

SECTION 9

PHYSICALLY PRINTING A REPORT

Triggering the printing of a report that has been written to a segment is straightforward. However, there are a few potential complications in obtaining the desired set of lines on a physical sheet of paper. Decisions must be made when the MRPG source program is written, when the MRPG-OS is invoked, and when the actual printout is triggered. These decisions interact, sometimes in non-obvious ways. This section is intended to assist the user in making best use of the available flexibility with a minimum number of MRPG/DPRINT experiments.

POTENTIAL PROBLEM AREAS

The following terminology is helpful in explaining the problems and in suggesting solutions.

Logical line
: A set of fields associated with the LINE keyword of the Line group.

Physical line
: A spatial area on the output media. For the line printers used at the central site, this area is usually 1/6 of an inch high by 136 columns wide. Sometimes, it is 1/8 of an inch high.

Logical page
: That collection of logical lines that the user wants to have printed as a unit, with no page heading and/or page footing lines except at the start and end of the unit.

Physical page
: A piece of paper, a portion of a microfiche, or a display screen on a terminal. For the line printers used at the central site, the piece of paper is usually 11 inches high by 14-7/8 inches wide. Other physical page sizes may also be used.

A logical page may be shorter, equal in length to, or longer than the physical page. The programmer's challenge is to specify the source program statements and dprint control arguments to produce a report whose layout is what the report's readers want.

Logical Versus Physical Lines

Usually, one logical line maps into one and only one physical line. However, it is possible, and at times may be desirable, for one logical line to occupy more than one physical line.

Suppose the output device is the central site line printer whose default column width is 136 columns. Logical lines that require 137, 138, 139, ..., 272 columns occupy two physical lines. Note that it is the quantity of columns required that is crucial. The character count may exceed 136 and still occupy only one physical line if sufficient overstriking occurs. This might be done to underline a heading. Each backspace and overstrike character consume zero column positions. Using a carriage return (015 octal) or many backspaces and the same text again to produce a darker line can yield a logical line that is much longer than 136 characters but that occupies only one physical line.

The current central site line printers have 136 printing positions, and 136 is the default line length value for the dprint command. The effective line length is shortened when the dprint -indent control argument is used. The dprint -line_length control argument can also change the effective line length. Thus, using the -indent and/or -line_length control arguments could decrease the effective physical length of a line such that a logical line now consumes more than one physical line. The "line overflow situation" is referred to in a later paragraph titled "Interaction Example".

Assume that the top portion of a physical page consists of:

Physical Line Numbers	Content
1-3	Three blank lines, for a top of page margin.
4,5	Two page heading lines.
6-8	Three blank lines for a separation between the page heading and the text.
9	The first detail line.

If, as is usually the case, lines 1-3 are skipped because a newpage character is sent to the printer and the Vertical Format Control (VFC) information and initial paper positioning is set up to cause skipping to line 4, then one could say that the first page heading line consumed four physical lines.

Further assume that the skipping of physical lines 6-8 is accomplished by specifying the MINLINE option with a value of 5 in the Report_Control group. In a sense, it can be said that the first detail logical line consumed four physical lines.

The same kind of physical line skipping may be done by specifying integer-14 or integer-15 in the Line group. These integers specify the absolute or relative line number that the defined line falls on. If the value of integer-14 is more than one higher than the line number for the previous line, or if integer-15 is greater than one, the logical line consumes more than one physical line.

Page Height

The choice of control arguments for the dprint command can affect the height of the page. The essence of these effects are discussed in the next few paragraphs. However, the writeup of the dprint command in the MPM Commands is the authoritative source of information and should be read to obtain the exact details.

Assume the central site line printer is set to six lines per inch and the paper is fan-folded every 11 inches. The number of lines that can be printed on a physical page depends on the information that has been loaded into the printer's VFC unit. If the standard system printer control values are not modified by the site and a segment that contains no newpage characters is printed with the dprint command but no control arguments, 60 lines are printed on each sheet of paper. The first one-half inch and the last one-half inch of paper are skipped because of the VFC data. However, printing that same segment with the -no_endpage control argument yields 63 lines on the first sheet and 66 lines on succeeding sheets.

The -page_length dprint control argument can change the effective page size from the VFC viewpoint into almost any quantity of lines per page.

The PAGELENGTH option of the Report_Control group can change the logical page size from the MRPG-OS viewpoint. If the PAGELENGTH value is set to 4, the report is sent to a segment, and the report is dprinted without control arguments, then only one line of the report is printed on each physical page.

INTERACTION EXAMPLE

To clarify the interrelationship between the MRPG Report_Control values and the dprint control arguments, consider the following example. Use the central site line printer. Set it to six lines per inch. Use blank paper that is fan-folded at eleven inch intervals. In order to save paper, since many copies of the report are needed, the report is photoreduced to 60% of its printed size before reproducing the report on eleven inch high paper. Thus, six lines per inch from the line printer are ten lines per inch in the reproduced report. With one inch top and bottom margins for the reproduced report, one eleven inch reproduced sheet has 90 report lines and represents 110 physical lines. Assume that two page heading lines and two page footing lines are wanted. Also assume that the "line overflow situation" discussed earlier in this section does not occur.

In the horizontal center of Figure 9-1, the numbers from 1 to 138 represent the physical line numbers. The ***** lines in that column of numbers represent the perforations in the paper.

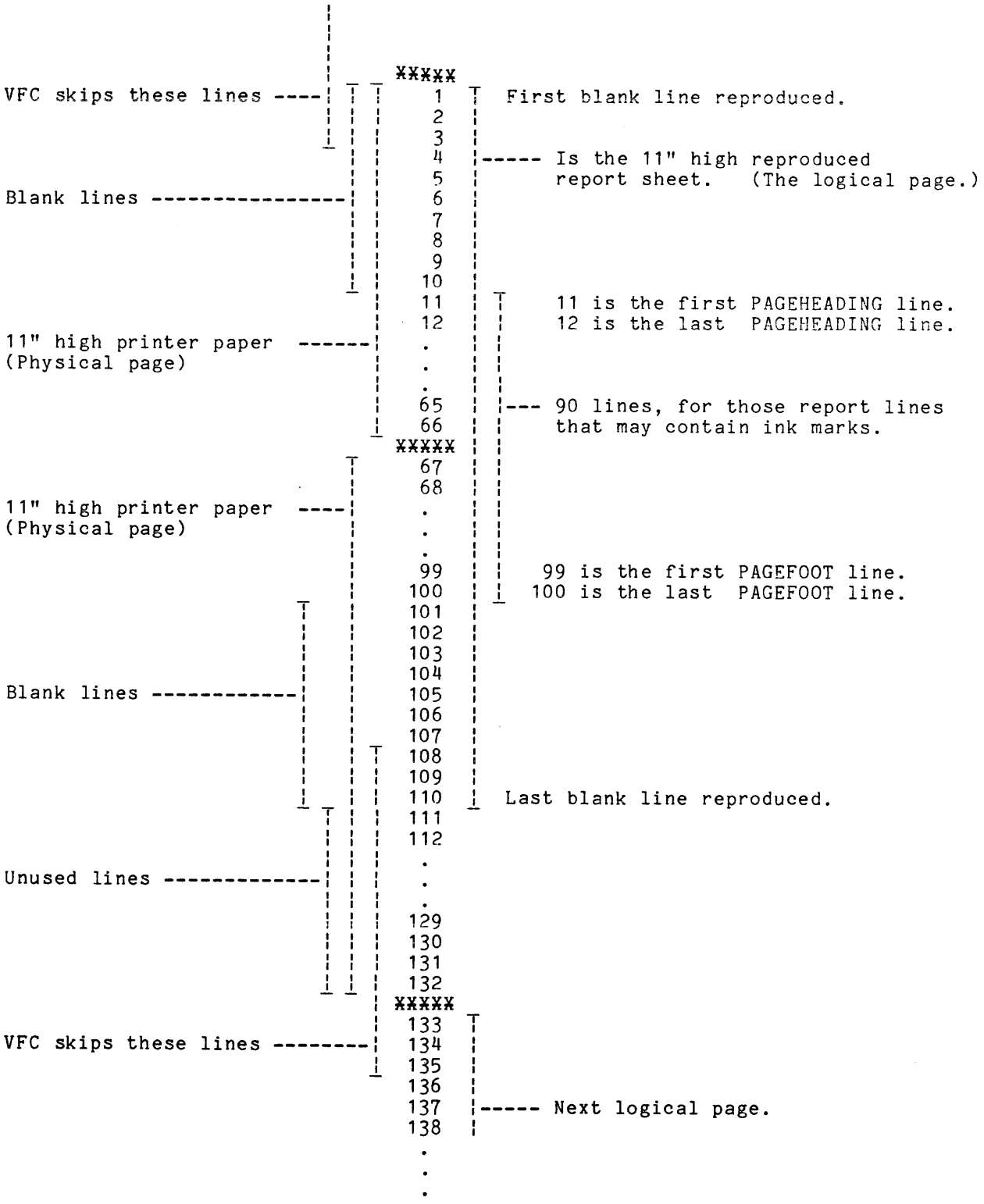


Figure 9-1. Physical Layout of a Long Logical Page

The crucial items for producing and printing a report with the vertical layout shown in Figure 9-1 are:

- In the MRPG source program

```
define 1 report your_report_name on "your_output"
pagelength 110
maxline 98      /* Last line number that detail lines can fall on. */
   .
   .
   .
2 pagehead,
 3 line 11 ... /* Define first page heading line. */
 3 line ...    /* Define last page heading line. */
   .
   .
   .
2 pagefoot,
 3 line 99 ... /* Define first page footing line. */
 3 line ...    /* Define second (and last) page footing line. */
   .
   .
   .
```

- In the dprint command line

```
dprint your_output -page_length 110 ...
```

LABEL LINES

Continue to use Figure 9-1 as an example. The dprint command has a few control arguments that can cause "label lines" to be printed. If "-top_label string_top" is specified with the dprint command, "string_top" is printed at the beginning of line 2 while specifying "-bottom_label string_bottom" prints "string_bottom" on line 109. Using "-label string_both" prints "string_both" on lines 2 and 109. The printing of these label lines does not affect the printing or spacing of any other lines.

Figure 9-1 presumed that the segment being printed had an Access Isolation Mechanism (AIM) access class name with a null value. This is the most common situation. Unless specific AIM actions are taken, processes and segments are at the AIM system_low level. The system default value for the system_low access class name is null. If the AIM is used to give a segment an access class name that is not null, e.g., COMPANY PROPRIETARY, then that access class name appears on lines 2 and 109 if the dprint label-type control arguments specify printing the access label.

APPENDIX A

THE RESERVED KEYWORDS

The table below shows every character string that is reserved as a keyword within the MRPG language including keywords that have short forms where applicable. In addition, all character strings for names that start with one uppercase letter followed by one underscore character are reserved for use as names in the generated PL/I source program.

KEYWORDS	
Long Form	Short Form
%DAY	
%HHMMSS	
%LEVEL	
%MMDDYY	
%MONTH	
%PAGENUMBER	
%REPEAT	
%SUBSTR	
%YYDDD	
ALIGN	
AND	&
ASCENDING	ASC
ATTACH	
BEGIN	
BEGINS	
BOOLEAN	BOOL
BREAK	
BSP	
CENTER	
CHARACTER	CHAR
COLUMN	COL
CONCATENATE	||
CONTAIN	
CONTAINS	
DECIMAL	DEC
DECLARE	DCL
DEFAULT	
DEFINE	
DELIMITED	

KEYWORDS	
Long Form	Short Form
DESCENDING	DESC
DETAIL	
DETAILFOOT	
DETAILHEAD	
DUPLICATE	DUPL
ELSE	
END	
ENDS	
EQ	=
FALSE	
FI	
FILE	
GE	=>
GT	>
HOLD	
IF	
IN	
INPUT	
KEY	
LEFT	
LET	
LE	<=
LINE	
LT	<
MAXLINE	MAXL
MINLINE	MINL
NE	^=
NO	
NOT	^

KEYWORDS	
Long Form	Short Form
ON	
OPTIONAL	
OR	|
PAGEFOOT	
PAGEHEAD	
PAGELENGTH	PGL
PAGEWIDTH	PGW
PARAMETER	PARM
PAUSE	
PICTURE	PIC
POSITION	
PRINT	
RECORD	
REPORT	
REPORTFOOT	
REPORTHEAD	
RIGHT	
SET	
SORT	
SPECIAL	
STREAM	
SWITCH	
TABLE	
THEN	
TRANSFORM	
TRUE	
VARYING	VAR
WORD	

APPENDIX B

ADDITIONAL SAMPLE PROGRAMS

This appendix contains several examples of MRPG programs. These programs illustrate specific points within the MRPG. In the real world, it would not be appropriate to write such simple MRPG programs. However, these examples are kept simple so that the primary feature in each example is not obscured.

Each example shows the input data used, the MRPG source program, and the output report(s) produced. Some examples also include a discussion. The examples include line numbers, to facilitate the discussion of specific points of interest. These line numbers were added to the input, program, or reports after the object programs executed. In those examples where the report(s) is written to a segment, the last character in the report segment is a newpage character (octal 014) which is printed in this appendix as the \014 string. There may also be newpage characters before the last line of the report. If so, they are shown as \014 strings. In general, reports in the examples are written to segments in the user's working directory. The pathname shown for the reports starts with [wd] which represents the Multics active function that returns the pathname of the working directory. Thus, [wd]>abcd refers to the segment named abcd in the working directory.

The outputs shown with examples are a copy of the actual reports produced by these sample programs when run on Multics.

The input data files and the MRPG source programs are supplied to customers as part of the MRPG software. Users may run these programs to verify that the MRPG produces the reports shown here. To do so, the user's working directory should be one in which the user can create and write segments. Type either of the following two lines:

```
archive  xf  >unbundled>mrpg_examples  run_mrpg_examples.ec
ac  xf  >unb>mrpg_examples  run_mrpg_examples.ec
```

The exec_com is extracted from the archive and written into the working directory. Then type either of the following two lines:

```
exec_com  run_mrpg_examples
ec  run_mrpg_examples
```

A list of the examples contained in the archive is displayed, along with a brief explanation of how to use the exec_com. Respond to the questions. The example selected is extracted from the archive, the PL/I program(s) is generated and compiled, and then the newly-compiled object program(s) is executed.

All of the segments extracted from the archive are established in the working directory, along with the generated PL/I segments, the compiled object segments, and those reports that are written into segments. The user is cautioned that extracting segments from the archive and running the examples create segments in the working directory. If there is any conflict with the names of already-existing segments, the user can create a new subdirectory, change to that new directory, and then extract the exec_com and run the examples.

TWO REPORTS

The Input (two_reports.mrpg.input in the archive)

```
Line
No.   The Actual Input Lines
----  ------------------------------------------------------------------------

 1    line 1 of two_reports.mrpg.input
 2    line 2 of two_reports.mrpg.input
```

The Source Program (two_reports.mrpg in the archive)

This program produces the two reports printed after this program from the input file shown above. This is a trivial example to show the essential steps required to produce more than one report. (Line 6 below, indicates "3 line 4". The output file places the data on the first line since MRPG assumes a dprint of the file where the data then appears on line 4.)

```
Line
No.   The Actual Source Program Lines
----  ------------------------------------------------------------------------

 1    /*  Simple program to produce two trivial reports.  */
 2    dcl 1 input stream file "two_reports.mrpg.input",
 3     2 the_data char(32);
 4    define 1 report report_one pagelength 12
 5      on file "two_reports.file_one.report",
 6     2 pagehead, 3 line 4, 4 "THIS REPORT PRODUCED ON " || %mmddyy, 3 line,
 7     2 detail detail_one,
 8      3 line +2, 4 "Line A, report one.  The input is ", 4 the_data,
 9      3 line,    4 "Line B, report one.  The input is ", 4 the_data;
10    define 1 report report_two pagelength 12
11      on file "two_reports.file_two.report",
12     2 pagehead, 3 line 4, 4 "THIS REPORT PRODUCED ON " || %mmddyy, 3 line,
13     2 detail detail_two,
14      3 line +2, 4 "Report two, line A.  The input is ", 4 the_data,
15      3 line,    4 "Report two, line B.  The input is ", 4 the_data;
16    begin() print report_one; print report_two;
17    end;
```

The Output

The report written into [wd]>two_reports.file_one.report is:

```
Line
No.   The Actual Output Lines
----  ------------------------------------------------------------------------

 1    THIS REPORT PRODUCED ON 02/17/78
 2
 3    Line A, report one.  The input is line 1 of two_reports.mrpg.input
 4    Line B, report one.  The input is line 1 of two_reports.mrpg.input
 5
 6    Line A, report one.  The input is line 2 of two_reports.mrpg.input
 7    Line B, report one.  The input is line 2 of two_reports.mrpg.input
 8    \014
```

The report written into [wd]>two_reports.file_two.report is:

```
Line
No.   The Actual Output Lines
----  ------------------------------------------------------------------------

 1    THIS REPORT PRODUCED ON 02/17/78
 2
 3    Report two, line A.  The input is line 1 of two_reports.mrpg.input
 4    Report two, line B.  The input is line 1 of two_reports.mrpg.input
 5
 6    Report two, line A.  The input is line 2 of two_reports.mrpg.input
 7    Report two, line B.  The input is line 2 of two_reports.mrpg.input
      \014
```

HOLD AND SORT

The Input (hold_and_sort.mrpg.input in the archive)

```
Line
No.   The Actual Input Lines
----  ------------------------------------------------------------------------

 1    duck    1  4  /* kind, in_stock, price */
 2    finch   4  2
 3    goose   1  3
 4    pigeon  2  4
 5    robin   2 10
```

The Source Program (hold_and_sort.mrpg in the archive)

This MRPG program reads and holds the input file, sorts the held file, and then uses the held "sorted file" to produce the report.

```
Line
No.   The Actual Source Program Lines
----  ------------------------------------------------------------------------

 1    /*  Simple example illustrating  hold  and  sort  */
 2    dcl 1 input stream file "hold_and_sort.mrpg.input",
 3         2 kind char(6), 2 in_stock dec(3), 2 price dec(3);
 4    dcl accum_value dec;
 5    define 1 report bird_value pagelength 14 on file "hold_and_sort.report",
 6     2 pagehead, 3 line 4, 4 "THIS REPORT PRODUCED ON " || %mmddyy, 3 line,
 7      3 line, 4 "Kind    In Stock  Price  Accum Value",
 8      3 line, 4 "------  --------  -----  -----------", 3 line,
 9     2 detail the_data, 3 line,
10       4 kind char(6) left, 4 in_stock char(10) right,
11       4 price char(7) right, 4 accum_value char(13) right
12          let (accum_value := accum_value + in_stock * price;);
13    begin () hold input;
14    begin (accum_value := 0;) sort in_stock desc, price asc;
15    print bird_value;   end;
```

The Output ([wd]>hold_and_sort.report)

```
Line
No.   The Actual Output Lines
----  ------------------------------------------------------------------------

 1    THIS REPORT PRODUCED ON 02/16/78
 2
 3    Kind    In Stock  Price  Accum Value
 4    ------  --------  -----  -----------
 5
 6    finch          4      2            8
 7    pigeon         2      4           16
 8    robin          2     10           36
 9    goose          1      3           39
10    duck           1      4           43
11    \014
```

BEGIN HOLD ASSIGN

The Input (begin_9_hold_assign.mrpg.input in the archive)

```
Line
No.   The Actual Input Lines
----  ------------------------------------------------------------------------

 1    11121314
 2    21222324
```

The Source Program (begin_8_hold_assign.mrpg in the archive)

This example illustrates the effects of placing assignment statements at various places within the source program. In some cases, the value resulting from the execution of an assignment statement is available during the current phase and also during later phases. In some cases, the result is available during only the current phase. In other cases, the result is not available at all. A discussion of specific cases follows the source program and the output reports.

This example includes one input file, two MRPG source programs, and four reports. The names of these seven segments are shown in the comments at the beginning of the MRPG source program. For brevity, let B_1_HA denote the begin_1_hold_assign version and let B_8_HA denote the begin_8_hold_assign version. Because the two source programs are almost identical, only one of them is included here. The major difference is that in B_8_HA a hold statement occurs in all eight phases, while in B_1_HA a hold statement occurs in only the first phase. The only other difference is in the segment names for the output reports, with "_1_" used for the B_1_HA reports and "_8_" used for the B_8_HA reports.

```
Line
No.   The Actual Source Program Lines
----  ------------------------------------------------------------------------

 1     /*  begin_1_hold_assign.mrpg   contains one hold statement, in phase one.
 2      *  begin_8_hold_assign.mrpg   contains eight holds, one per phase.
 3      *  These two MRPG programs illustrate the interactions between:
 4      *      -- hold statements
 5      *      -- assignment statements inside begin parentheses
 6      *      -- assignment statements in execute loop
 7      *  For both programs, the input file is  begin_9_hold_assign.mrpg.input
 8      *  "in_" denotes input field; "lv_" denotes local variable.
 9      *  Report names for _1_ version are   begin_1_hold_assign.(in lv).report
10      *  Report names for _8_ version are   begin_8_hold_assign.(in lv).report
11      */
12     dcl 1 input stream file "begin_9_hold_assign.mrpg.input",
13      2 in_1 dec(2), 2 in_2 dec(2), 2 in_3 dec(2), 2 in_4 dec(2);
14     dcl lv_1 dec;  dcl lv_2 dec;  dcl lv_3 dec;  dcl lv_4 dec;
15     dcl phase dec;  dcl input_record_number dec;
16     define 1 report in_report on file "begin_8_hold_assign.in.report",
17     2 detail in_data_line, 3 line,
18       4 "Phase ", 4 phase, 4 "- Record ", 4 input_record_number,
19       4 "   in_1  = ", 4 in_1,   4 "   in_2  = ", 4 in_2,
20       4 "   in_3  = ", 4 in_3,   4 "   in_4  = ", 4 in_4,
21      3 line if (input_record_number = 2);     /* Blank line between phases. */
22     define 1 report lv_report on file "begin_8_hold_assign.lv.report",
23     2 detail lv_data_line, 3 line,
24       4 "Phase ", 4 phase, 4 "- Record ", 4 input_record_number,
25       4 "   lv_1  = ", 4 lv_1,   4 "   lv_2  = ", 4 lv_2,
26       4 "   lv_3  = ", 4 lv_3,   4 "   lv_4  = ", 4 lv_4,
27      3 line if (input_record_number = 2);     /* Blank line between phases. */
28
```

```
29
30      /* ----- PHASE 1 ----- */
31      begin (phase := 1; input_record_number := 0;
32             lv_1 := 85; lv_2 := 86; lv_3 := 87; lv_4 := 88;)
33
34      input_record_number := input_record_number + 1;
35      print in_report;  print lv_report;
36      hold in_1, in_2, lv_1, lv_2;      /* Same in _1_ and _8_ versions. */
37
38
39      /* ----- PHASE 2 ----- */
40      begin (phase := phase + 1; input_record_number := 0;)
41
42      input_record_number := input_record_number + 1;
43      print in_report;  print lv_report;
44      hold;  /* This "hold;" statement is commented out in the
45              * _1_ version of this program.  */
46
47
48      /* ----- PHASE 3 ----- */
49      begin (phase := phase + 1; input_record_number := 0;
50             in_1 := in_1 + 1; in_3 := in_3 + 1;
51             lv_1 := lv_1 + 1; lv_3 := lv_3 + 1;)
52
53      input_record_number := input_record_number + 1;
54      print in_report;  print lv_report;
55      hold;  /* This "hold;" statement is commented out in the
56              * _1_ version of this program.  */
57
58
59      /* ----- PHASE 4 ----- */
60      begin (phase := phase + 1; input_record_number := 0;)
61
62      input_record_number := input_record_number + 1;
63      print in_report;  print lv_report;
64      hold;  /* This "hold;" statement is commented out in the
65              * _1_ version of this program.  */
66
67
68      /* ----- PHASE 5 ----- */
69      begin (phase := phase + 1; input_record_number := 0;)
70
71      input_record_number := input_record_number + 1;
72      in_2 := in_2 + 1; in_4 := in_4 + 1;
73      lv_2 := lv_2 + 1; lv_4 := lv_4 + 1;
74      print in_report;  print lv_report;
75      hold;  /* This "hold;" statement is commented out in the
76              * _1_ version of this program.  */
77
78
79      /* ----- PHASE 6 ----- */
80      begin (phase := phase + 1; input_record_number := 0;)
81
82      input_record_number := input_record_number + 1;
83      print in_report;  print lv_report;
84      hold;  /* This "hold;" statement is commented out in the
85              * _1_ version of this program.  */
86
```

```
 87
 88    /* ----- PHASE 7 ----- */
 89    begin (phase := phase + 1; input_record_number := 0;
 90           in_1 := in_1 + 1; in_3 := in_3 + 1;
 91           lv_1 := lv_1 + 1; lv_3 := lv_3 + 1;)
 92
 93    input_record_number := input_record_number + 1;
 94    in_1 := in_1 + 1; in_3 := in_3 + 1;
 95    lv_1 := lv_1 + 1; lv_3 := lv_3 + 1;
 96    print in_report;  print lv_report;
 97    hold;  /* This "hold;" statement is commented out in the
 98            * _1_ version of this program.  */
 99
100
101    /* ----- PHASE 8 ----- */
102    begin (phase := phase + 1; input_record_number := 0;)
103
104    input_record_number := input_record_number + 1;
105    print in_report;  print lv_report;
106    hold;  /* This "hold;" statement is commented out in the
107            * _1_ version of this program.  */
108    end;
```

The Output

The four reports are printed in this sequence:

```
begin_8_hold_assign.in.report
begin_1_hold_assign.in.report
begin_8_hold_assign.lv.report
begin_1_hold_assign.lv.report
```

```
Line
No.   The Actual Output Lines For  begin_8_hold_assign.in.report
----  ------------------------------------------------------------------

 1    Phase 1 - Record 1    in_1  = 11    in_2  = 12    in_3  = 13    in_4  = 14
 2    Phase 1 - Record 2    in_1  = 21    in_2  = 22    in_3  = 23    in_4  = 24
 3
 4    Phase 2 - Record 1    in_1  = 11    in_2  = 12    in_3  = 23    in_4  = 24
 5    Phase 2 - Record 2    in_1  = 21    in_2  = 22    in_3  = 23    in_4  = 24
 6
 7    Phase 3 - Record 1    in_1  = 11    in_2  = 12    in_3  = 24    in_4  = 24
 8    Phase 3 - Record 2    in_1  = 21    in_2  = 22    in_3  = 24    in_4  = 24
 9
10    Phase 4 - Record 1    in_1  = 11    in_2  = 12    in_3  = 24    in_4  = 24
11    Phase 4 - Record 2    in_1  = 21    in_2  = 22    in_3  = 24    in_4  = 24
12
13    Phase 5 - Record 1    in_1  = 11    in_2  = 13    in_3  = 24    in_4  = 25
14    Phase 5 - Record 2    in_1  = 21    in_2  = 23    in_3  = 24    in_4  = 26
15
16    Phase 6 - Record 1    in_1  = 11    in_2  = 13    in_3  = 24    in_4  = 26
17    Phase 6 - Record 2    in_1  = 21    in_2  = 23    in_3  = 24    in_4  = 26
18
19    Phase 7 - Record 1    in_1  = 12    in_2  = 13    in_3  = 26    in_4  = 26
20    Phase 7 - Record 2    in_1  = 22    in_2  = 23    in_3  = 27    in_4  = 26
21
22    Phase 8 - Record 1    in_1  = 12    in_2  = 13    in_3  = 27    in_4  = 26
23    Phase 8 - Record 2    in_1  = 22    in_2  = 23    in_3  = 27    in_4  = 26
24
25    \014
```

```
Line
No.   The Actual Output Lines For  begin_1_hold_assign.in.report
----  ------------------------------------------------------------------

 1    Phase 1 - Record 1    in_1  = 11    in_2  = 12    in_3  = 13    in_4  = 14
 2    Phase 1 - Record 2    in_1  = 21    in_2  = 22    in_3  = 23    in_4  = 24
 3
 4    Phase 2 - Record 1    in_1  = 11    in_2  = 12    in_3  = 23    in_4  = 24
 5    Phase 2 - Record 2    in_1  = 21    in_2  = 22    in_3  = 23    in_4  = 24
 6
 7    Phase 3 - Record 1    in_1  = 11    in_2  = 12    in_3  = 24    in_4  = 24
 8    Phase 3 - Record 2    in_1  = 21    in_2  = 22    in_3  = 24    in_4  = 24
 9
10    Phase 4 - Record 1    in_1  = 11    in_2  = 12    in_3  = 24    in_4  = 24
11    Phase 4 - Record 2    in_1  = 21    in_2  = 22    in_3  = 24    in_4  = 24
12
13    Phase 5 - Record 1    in_1  = 11    in_2  = 13    in_3  = 24    in_4  = 25
14    Phase 5 - Record 2    in_1  = 21    in_2  = 23    in_3  = 24    in_4  = 26
15
16    Phase 6 - Record 1    in_1  = 11    in_2  = 12    in_3  = 24    in_4  = 26
17    Phase 6 - Record 2    in_1  = 21    in_2  = 22    in_3  = 24    in_4  = 26
18
19    Phase 7 - Record 1    in_1  = 12    in_2  = 12    in_3  = 26    in_4  = 26
20    Phase 7 - Record 2    in_1  = 22    in_2  = 22    in_3  = 27    in_4  = 26
21
22    Phase 8 - Record 1    in_1  = 11    in_2  = 12    in_3  = 27    in_4  = 26
23    Phase 8 - Record 2    in_1  = 21    in_2  = 22    in_3  = 27    in_4  = 26
24
25    \014
```

```
Line
No.   The Actual Output Lines For  begin_8_hold_assign.lv.report
----  ------------------------------------------------------------------------

 1    Phase 1 - Record 1    lv_1  = 85    lv_2  = 86    lv_3  = 87    lv_4  = 88
 2    Phase 1 - Record 2    lv_1  = 85    lv_2  = 86    lv_3  = 87    lv_4  = 88
 3
 4    Phase 2 - Record 1    lv_1  = 85    lv_2  = 86    lv_3  = 87    lv_4  = 88
 5    Phase 2 - Record 2    lv_1  = 85    lv_2  = 86    lv_3  = 87    lv_4  = 88
 6
 7    Phase 3 - Record 1    lv_1  = 85    lv_2  = 86    lv_3  = 88    lv_4  = 88
 8    Phase 3 - Record 2    lv_1  = 85    lv_2  = 86    lv_3  = 88    lv_4  = 88
 9
10    Phase 4 - Record 1    lv_1  = 85    lv_2  = 86    lv_3  = 88    lv_4  = 88
11    Phase 4 - Record 2    lv_1  = 85    lv_2  = 86    lv_3  = 88    lv_4  = 88
12
13    Phase 5 - Record 1    lv_1  = 85    lv_2  = 87    lv_3  = 88    lv_4  = 89
14    Phase 5 - Record 2    lv_1  = 85    lv_2  = 87    lv_3  = 88    lv_4  = 90
15
16    Phase 6 - Record 1    lv_1  = 85    lv_2  = 87    lv_3  = 88    lv_4  = 90
17    Phase 6 - Record 2    lv_1  = 85    lv_2  = 87    lv_3  = 88    lv_4  = 90
18
19    Phase 7 - Record 1    lv_1  = 86    lv_2  = 87    lv_3  = 90    lv_4  = 90
20    Phase 7 - Record 2    lv_1  = 86    lv_2  = 87    lv_3  = 91    lv_4  = 90
21
22    Phase 8 - Record 1    lv_1  = 86    lv_2  = 87    lv_3  = 91    lv_4  = 90
23    Phase 8 - Record 2    lv_1  = 86    lv_2  = 87    lv_3  = 91    lv_4  = 90
24
25    \014
```

```
Line
No.   The Actual Output Lines For  begin_1_hold_assign.lv.report
----  ------------------------------------------------------------------------

 1    Phase 1 - Record 1    lv_1  = 85    lv_2  = 86    lv_3  = 87    lv_4  = 88
 2    Phase 1 - Record 2    lv_1  = 85    lv_2  = 86    lv_3  = 87    lv_4  = 88
 3
 4    Phase 2 - Record 1    lv_1  = 85    lv_2  = 86    lv_3  = 87    lv_4  = 88
 5    Phase 2 - Record 2    lv_1  = 85    lv_2  = 86    lv_3  = 87    lv_4  = 88
 6
 7    Phase 3 - Record 1    lv_1  = 85    lv_2  = 86    lv_3  = 88    lv_4  = 88
 8    Phase 3 - Record 2    lv_1  = 85    lv_2  = 86    lv_3  = 88    lv_4  = 88
 9
10    Phase 4 - Record 1    lv_1  = 85    lv_2  = 86    lv_3  = 88    lv_4  = 88
11    Phase 4 - Record 2    lv_1  = 85    lv_2  = 86    lv_3  = 88    lv_4  = 88
12
13    Phase 5 - Record 1    lv_1  = 85    lv_2  = 87    lv_3  = 88    lv_4  = 89
14    Phase 5 - Record 2    lv_1  = 85    lv_2  = 87    lv_3  = 88    lv_4  = 90
15
16    Phase 6 - Record 1    lv_1  = 85    lv_2  = 86    lv_3  = 88    lv_4  = 90
17    Phase 6 - Record 2    lv_1  = 85    lv_2  = 86    lv_3  = 88    lv_4  = 90
18
19    Phase 7 - Record 1    lv_1  = 86    lv_2  = 86    lv_3  = 90    lv_4  = 90
20    Phase 7 - Record 2    lv_1  = 86    lv_2  = 86    lv_3  = 91    lv_4  = 90
21
22    Phase 8 - Record 1    lv_1  = 85    lv_2  = 86    lv_3  = 91    lv_4  = 90
23    Phase 8 - Record 2    lv_1  = 85    lv_2  = 86    lv_3  = 91    lv_4  = 90
24
25    \014
```

Discussion

The results for in_3, in_4, lv_3, and lv_4 are identical for B_1_HA and B_8_HA in all eight phases because none of these fields or variables are included in the hold statement in phase 1 on program line 36.

In begin_8_hold_assign.in.report the value of in_3 remains as 23 on lines 4 and 5 because 23 is the value of in_3 in the second input record. The value of in_3 changes to 24 on report line 7 because of program line 50. Program line 90 changes the value from the 24 on report line 17 to 25 before the first input record is made available. Program line 94 changes the value to 26 on report line 19 when the first of the held input records is made available and to 27 on report line 20 when the second record is made available.

Similar changes to the values for in_4, lv_3, and lv_4 occur because of similar placements of assignment statements for those items. Most of the results for in_1, in_2, lv_1, and lv_2 are also due to similar placements of assignment statements for these items.

The values of in_1 in phase 8 differ between B_1_HA and B_8_HA because of the hold statement on program line 97. The hold in B_8_HA causes the changed value of in_1 to be retained for use in later phases. Since there is no corresponding hold statement in B_1_HA, the results of the incrementing of in_1 disappear at the end of phase 7, and the values for phase 8 are the values that were held during phase 1. The same effect is observed for in_2 when going from phase 4 to phase 5. However, in phases 6 through 8 of B_8_HA, the values of in_2 are those that were established in phase 5. These values are carried over to phases 6 through 8 because of the hold statement in phase 5 on program line 75.

The assignment statements for in_1 and lv_1 on program lines 50 and 51 have no effect on the reports because in_1 and lv_1 are held items. The incrementing is performed on some leftover values, but those leftover values are overwritten when the first held input record becomes available.

APPENDIX C

THE report_ I/O MODULE

INTRODUCTION

The report_ procedure is an I/O module, in the same sense that tty_ and vfile_ are I/O modules. However, because the application of report_ is closely connected to MRPG, the writeup of report_ is found in this manual, rather than in the MPM Subroutines.

This appendix contains a description of the report_ I/O module, using the same format as is used for the I/O module descriptions in the MPM Subroutines. Following the report_ description is a discussion of how the report_ I/O module interacts with other procedures in a Multics system.

Name: report_

This I/O module provides a mechanism for supplying input data to the report generation portion of an MRPG-OS. The implementation almost completely isolates the MRPG-OS from the details of I/O switches and modules. Should changes be made in the future to the general I/O switch approach, and/or to the content of I/O control blocks, the report_ I/O module can be changed and existing MRPG-OS should be able to continue producing the same reports.

Entries in the module are not called directly by users; rather, the module is accessed through the I/O system. (See "Multics Input/Output System" and "File Input/Output" in Section 5 of the MPM Reference Guide for a general description of the I/O system and a discussion of files, respectively.)

Attach Description

The attach description has the following form:

```
report_ ref_name {parameter list}
```

where:

1. ref_name
   is the reference name used when the MRPG-OS is initiated.

2. parameter list
   is a list of parameter values required by the MRPG-OS. The items in the list are separated by spaces. The sequence of the items must match the sequence required by the MRPG-OS.

Open Operation

The following opening modes are supported:

```
stream_output
sequential_output
```

An existing file is truncated to zero.

Only write access is required on the file.

Writing Operations

These writing operations are supported for these opening modes:

stream_output	supports	put_chars
sequential_output	supports	write_record

No other writing operations are supported.

Other Operations

These operations are supported:

close
detach_iocb

These operations are not supported:

all read-type operations
all key-type operations
delete_record
position
modes
control

File Position Designators

The standard file position designators:

next byte
next record
current record
key for insertion

are not used by the report_ I/O module.

INTERACTION WITH OTHER PROCEDURES

The remainder of this appendix provides a summary of how the report_ I/O module interacts with the MRPG-OS and with an input data supplier such as LINUS.

Gross Structure of an MRPG-OS

An MRPG-OS can be thought of as consisting of two parts, an input data supplier part and a report production part. For brevity, these two parts are referred to as MRPG-OS-input and MRPG-OS-report.

When the MRPG-OS is invoked as a command, the MRPG-OS-input obtains the input data.

When the MRPG-OS is used as an I/O appendage, the input data is obtained by some external procedure, such as LINUS.

In both cases, the input data is passed through report_ to MRPG-OS-report, which produces the report or reports.

The Input Data Supplier Part

The Declare_Input_File group includes the FILE and ATTACH keywords. If either is specified, it is possible for the MRPG-OS-input to be invoked as a command and to obtain the input data. It is also possible for an external procedure to obtain the input data. If the MRPG-OS is used as an I/O appendage, the FILE or ATTACH information and associated PL/I statements are ignored.

If neither FILE nor ATTACH are specified, the input data must be obtained by an external procedure. It is impossible to produce the reports by invoking the MRPG-OS as a command.

The Report Production Part

This part of the MRPG-OS converts each input data line/record into one or more report lines/records in one or more reports. These actions are independent of the origin of the input data.

Every report is written to either a file or a switch. The MRPG-OS-report calls on the iox_ subroutine to attach, open, write to, close, and detach the report files and switches.

The only external entry point used by MRPG-OS-input or an external procedure (e.g., LINUS) is:

report_attach

A Command Scenario

In this scenario, the MRPG-OS is invoked as a command. The input file is in a segment and the single output report is written to a segment. Each input line produces one report line.

For purposes of referring to the following steps in the next scenario, call this scenario A.

A1. The MRPG-OS is invoked from command level.

A2. MRPG-OS-input calls on iox_, report_, and MRPG-OS-report to set up a report switch for delivering data to MRPG-OS-report.

A3. MRPG-OS-input attaches and opens the input file.

A4. MRPG-OS-input obtains the next data line. When there is no more data, go to step A11.

A5. MRPG-OS-input writes one data line via iox_.

A6. Information as to the location, length, etc., of that data line proceeds from iox_ through report_ to MRPG-OS-report.

A7. The first time control reaches here, MRPG-OS-report attaches and opens the report file using the vfile_ I/O module.

A8. MRPG-OS-report manipulates its input data line to produce one report line. In more complicated situations, report, page, detail heading and footing lines, and multiple data lines are produced when appropriate as part of this step for one or more reports. If needed for second or subsequent phases, the input data and other data is held during this step.

A9. MRPG-OS-report writes the report line.

A10. MRPG-OS-report returns control through report_ and iox_ to MRPG-OS-input which loops back to step A4.

A11. MRPG-OS-input closes and detaches the input file.

A12. MRPG-OS-input closes the report switch set up in step A2.

A13. As part of accomplishing the previous step, control passes through iox_ and report_ to MRPG-OS-report. If the MRPG-OS program contains more than one phase, the second and all subsequent phases are executed at this time. The report file set up in step A7 is closed and detached. Control returns to MRPG-OS-input through report_ and iox_.

A14. MRPG-OS-input detaches the report switch closed in step A12.

A15. MRPG-OS-input returns control to the command processor.

An I/O Appendage Scenario

In this scenario, the MRPG-OS is used as an I/O appendage by some external procedure. For brevity, LINUS is assumed to be that external procedure. Otherwise, this scenario is like scenario A.

1. LINUS is invoked from command level.

2. The desired data base is selected and made available. The desired LILA requests are set up to produce the data lines of interest. A request of this form is issued:

   report <pathname of the MRPG-OS>

3. Similar to A2. LINUS calls on iox_, report_, and MRPG-OS-report to set up a report switch for delivering data to MRPG-OS-report.

4. Same function as A4. LINUS builds its next data line. When there is no more data, go to step 11.

5. Similar to A5. LINUS writes one data line via iox_.

6. Identical to A6. Information as to the location, length, etc., of that data line proceeds from iox_ through report_ to MRPG-OS-report.

7. Identical to A7. The first time control reaches here, MRPG-OS-report attaches and opens the report file using the vfile_ I/O module.

8. Identical to A8. MRPG-OS-report manipulates its input data line to produce one report line. In more complicated situations, report, page, detail heading and footing lines, and multiple data lines are produced when appropriate as part of this step for one or more reports. If needed for second or subsequent phases, the input date and other data is held during this step.

9. Identical to A9. MRPG-OS-report writes the report line.

10. Similar to A10. MRPG-OS-report returns control through report_ and iox_ to LINUS which loops back to step 4.

11. Similar to A12. Since the report request made in step 2 has been carried out, LINUS closes the report switch set up in step 3.

12. Similar to A13. As part of accomplishing the previous step, control passes through iox_ and report_ to MRPG-OS-report. If the MRPG-OS program contains more than one phase, the second and all subsequent phases are executed at this time. The report file set up in step 7 is closed and detached. Control returns through report_ and iox_ to the MRPG-OS-input which returns control to LINUS.

13. Similar to A14. LINUS detaches the report switch closed in step 11.

14. Similar to A11. If the user of LINUS so requests, LINUS closes and detaches the connections to the data base.

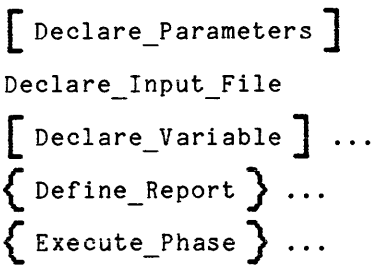
15. Similar to A15. Assuming the user is done and so requests, LINUS returns control to the command processor.

APPENDIX D

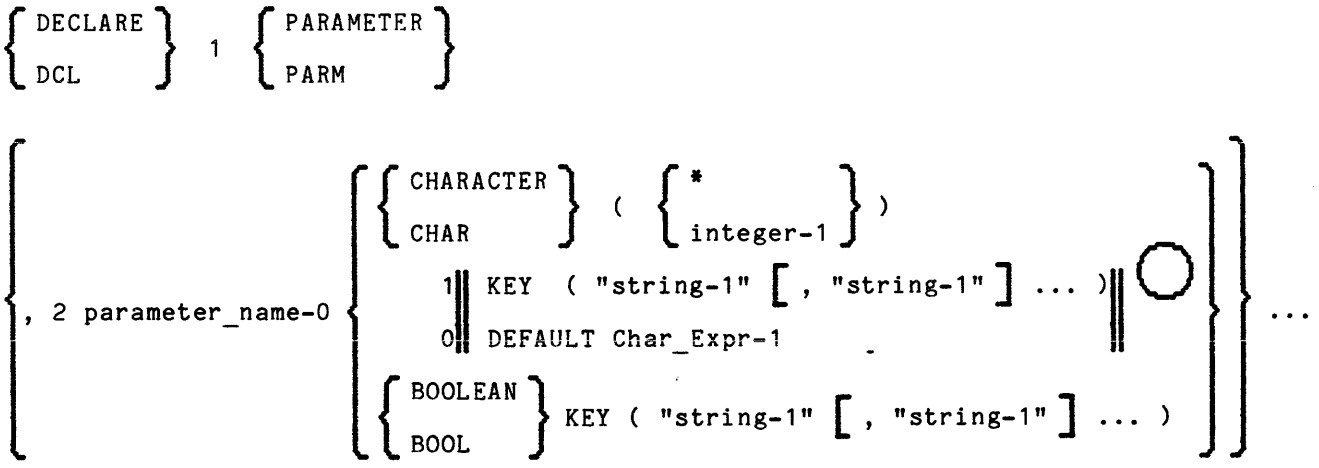
GENERAL FORMAT DIAGRAMS

This section consists of the general format diagrams that appear in Section 5. They are collected in this appendix so that the knowledgeable user may easily refer to any of the diagrams without having to flip back and forth as is required in Section 5 because the groups there are arranged in alphabetical order. However, in this appendix, the groups are arranged in their hierarchial order, which generally corresponds to the sequence in which the major groups must occur in a program.

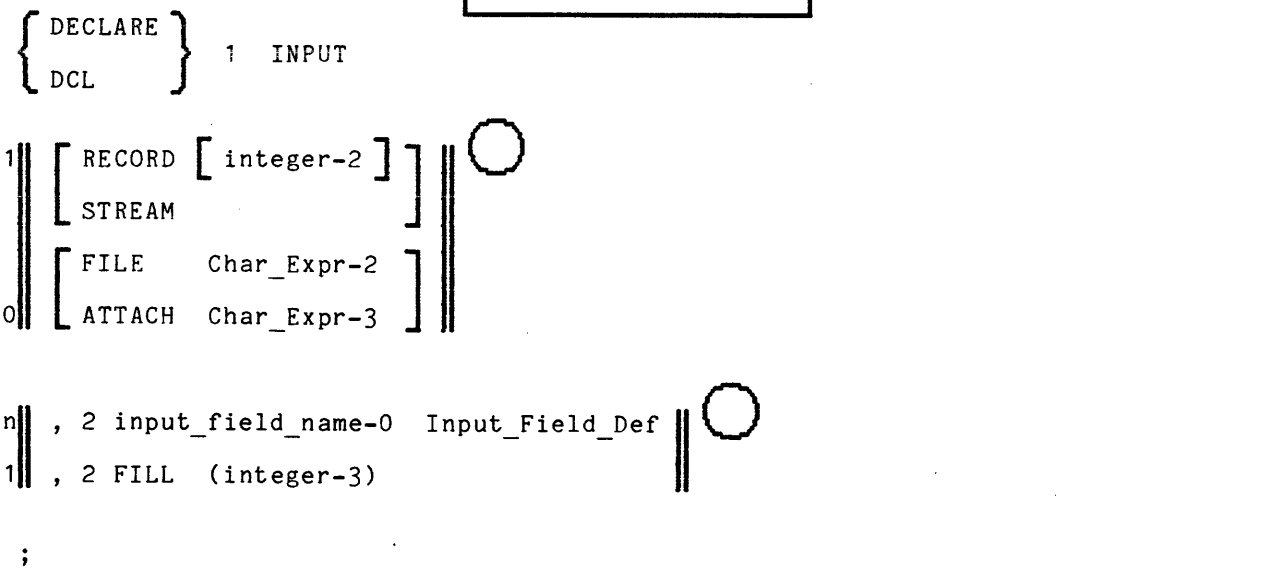
The_MRPG_Program



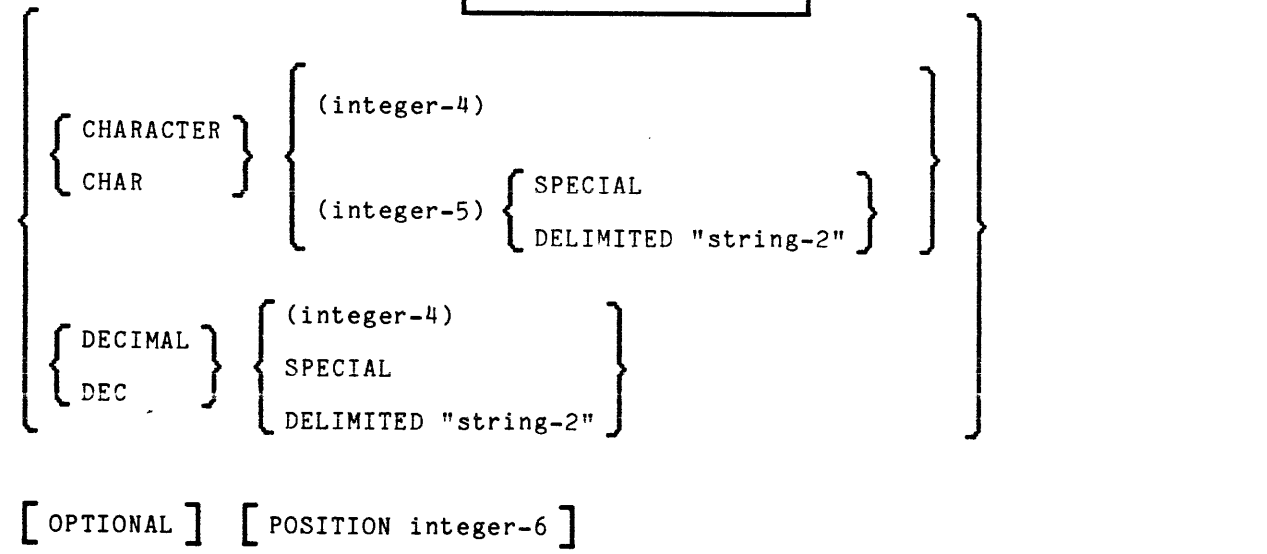
Declare_Parameters



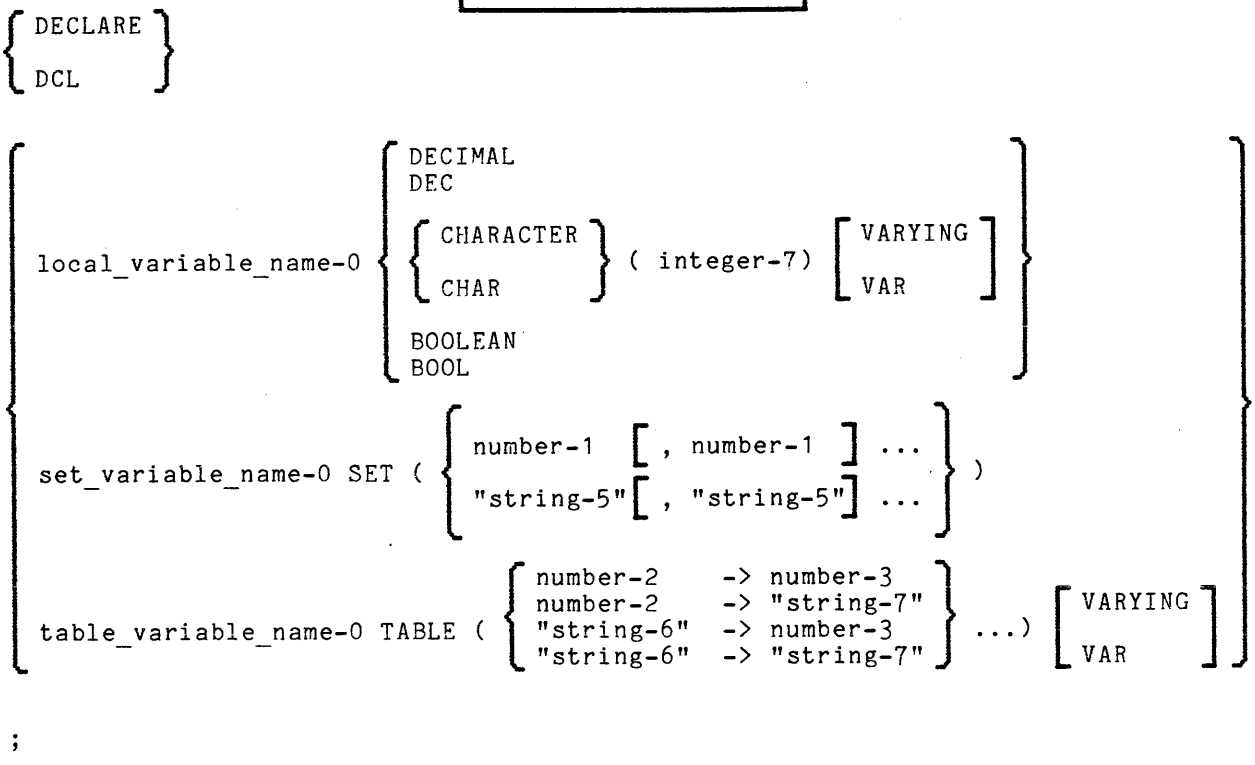
Declare_Input_File



Input_Field_Def

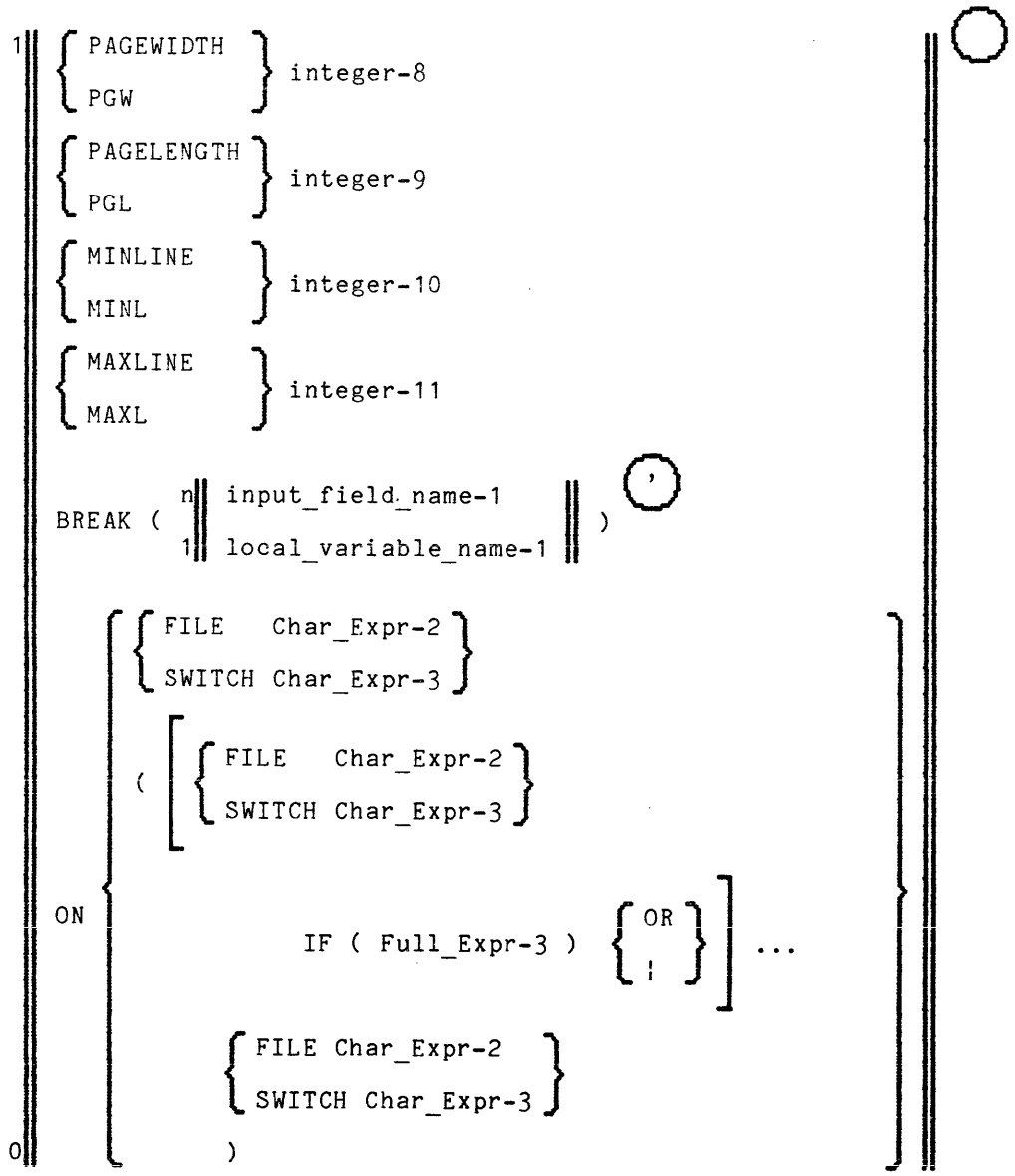


Declare_Variable

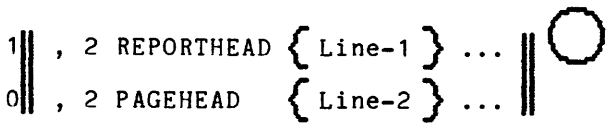


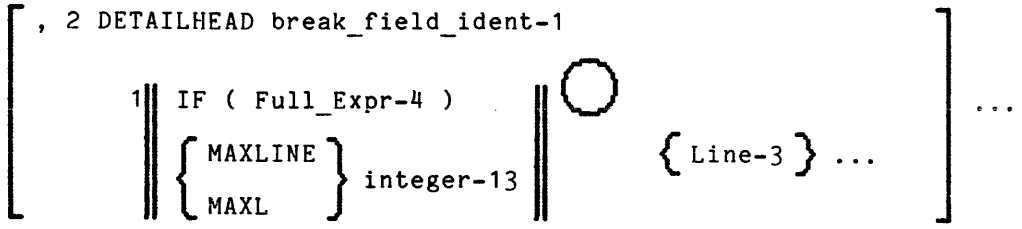
```
DEFINE 1 REPORT report_name-0
[ Report_Control ]
[ Heading ]
{ Detail } ...
[ Footing ]
;
```

Report_Control

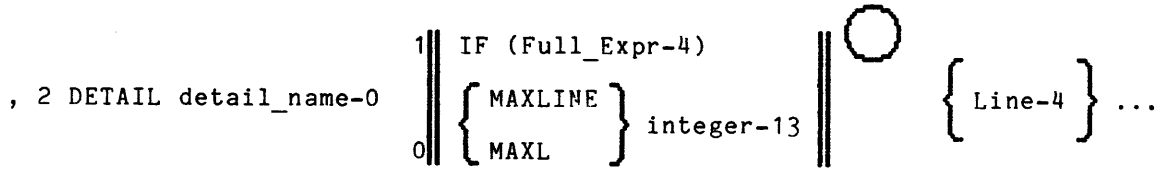


Heading

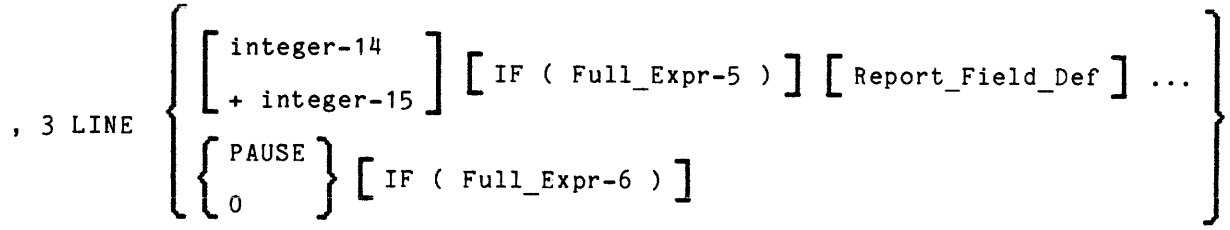




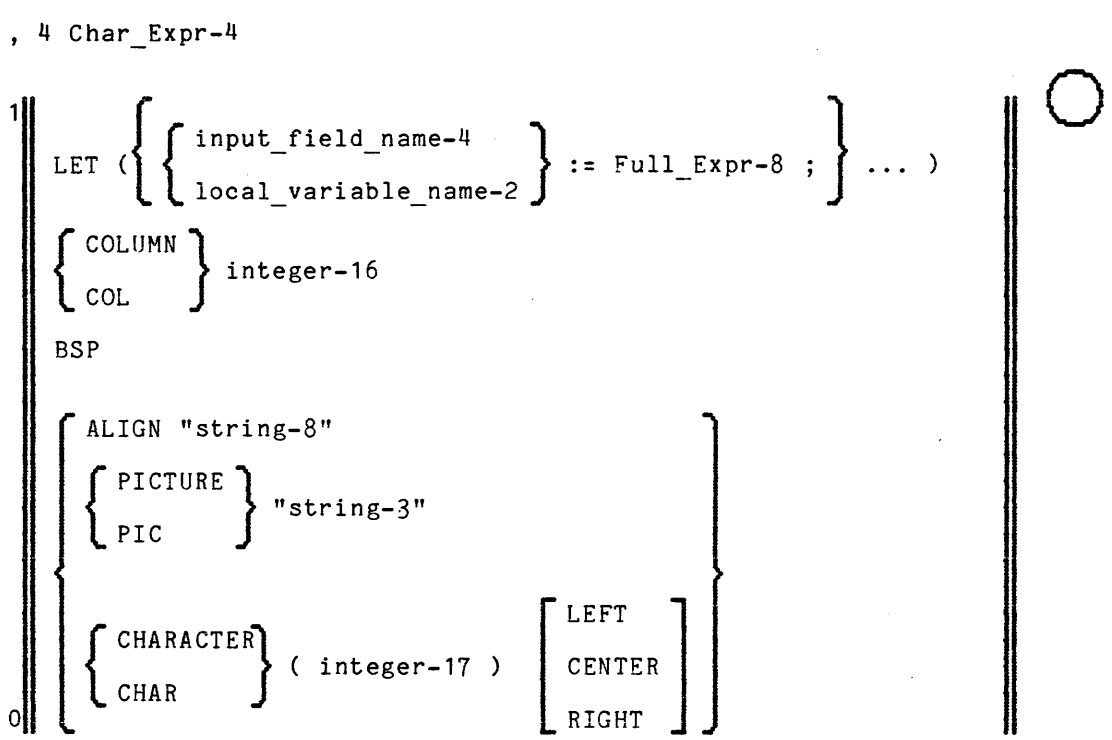
Detail



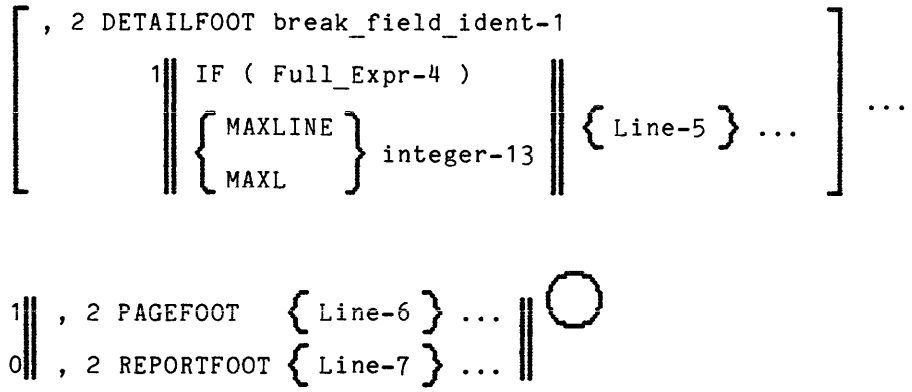
Line



Report_Field_Def



Footing



Execute_Phase

Format 1: (Valid only for the first phase.)

```
BEGIN ( [ local_variable_name-2 := Full_Expr-9 ; ] ... )

                             ┌    n║ INPUT                    ║ (,)   ┐
[ Loop_Statement ] ...       │ HOLD║ input_field_name-4       ║     ; │
                             └    0║ local_variable_name-3    ║       ┘
```

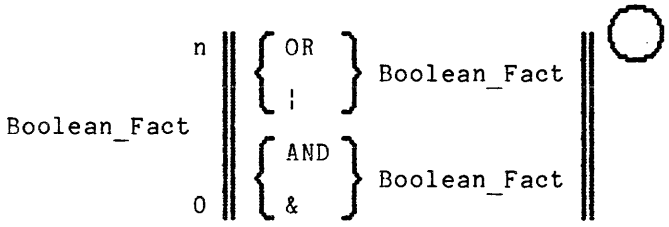
Format 2: (Valid for all phases after the first phase.)

```
BEGIN ( [ local_variable_name-2 := Full_Expr-9 ; ] ... )

┌      n║ { input_field_name-5     } [ ASCENDING  ]║ (,)  [    { DUPLICATE } ]   ┐
│ SORT  ║ {                        } [ ASC        ]║      [ NO {           } ] ; │
└      1║ { local_variable_name-4  } [ DESCENDING ]║      [    { DUPL      } ]   ┘
                                     [ DESC       ]

{ Loop_Statement } ... [ HOLD ; ]
```

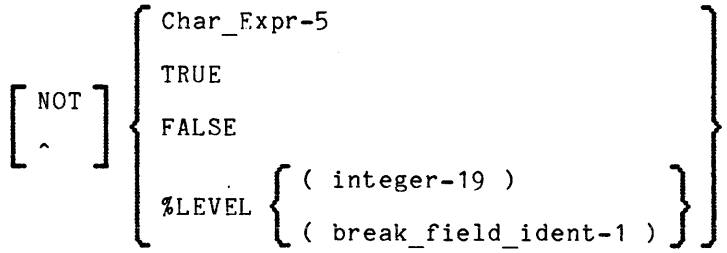
where Loop_Statement is:

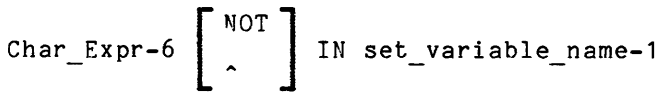
```
{ { input_field_name-4    }                                      }
{ { local_variable_name-2 } := Full_Expr-8 ;                     }
{                                                                }
{ PRINT { report_name-1 } ;                                      }
{       { detail_name-1 }                                        }
{                                                                }
{ IF Full_Expr-10 THEN { Loop_Statement } ...                    }
{            [ ELSE { Loop_Statement } ... ]   FI ;              }
{ ;                                                              }
```

Full_Expr



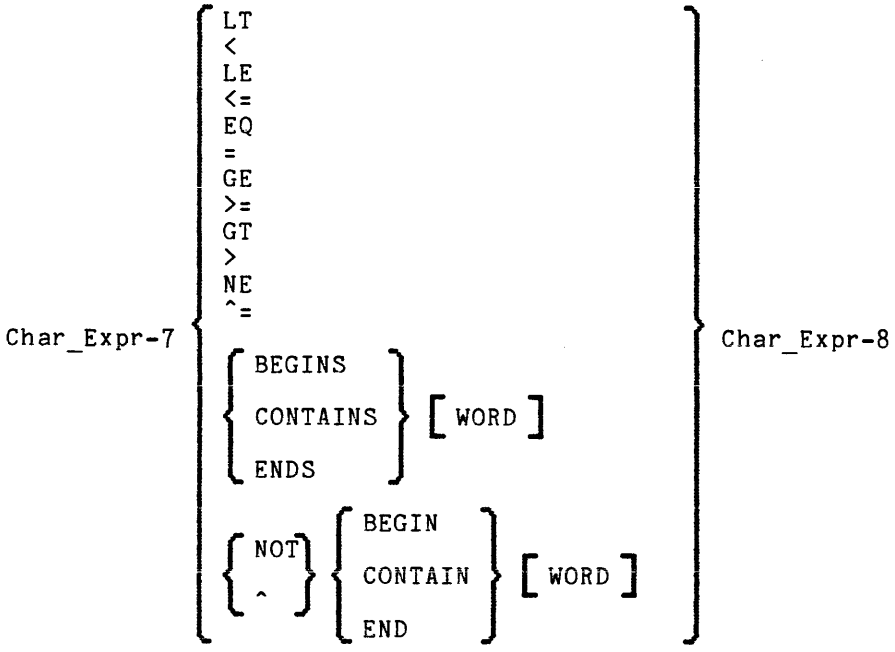
where Boolean_Fact is



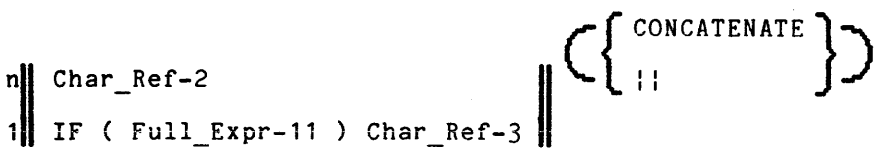


Relationship_Test

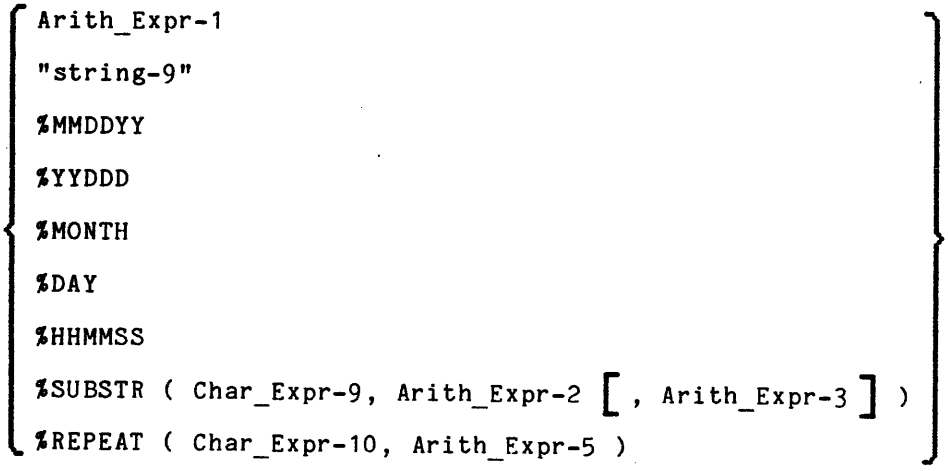
Relationship_Test



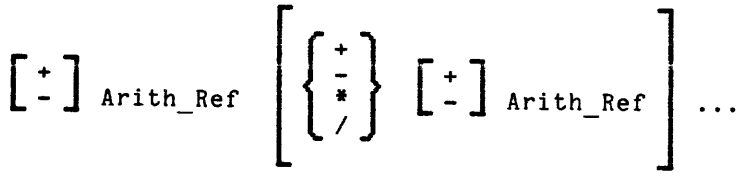
Char_Expr



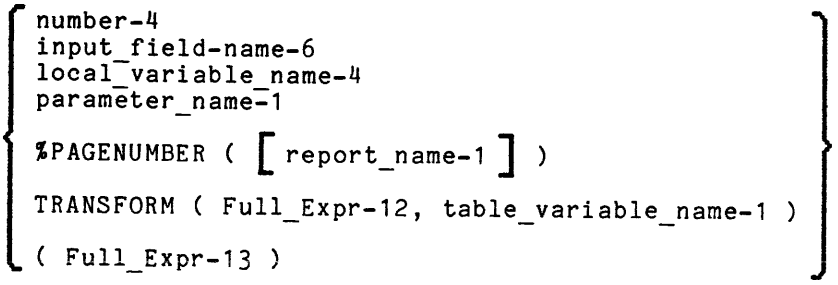
Char_Ref



Arith_Expr



where Arith_Ref is



MULTICS REPORT PROGRAM GENERATOR (MRPG) REFERENCE MANUAL ADDENDUM A

SUBJECT

Additions and Changes to the Multics Report Program Generator Reference Manual

SPECIAL INSTRUCTIONS

This is the first addendum to CC69, Revision 0 dated March 1978. Insert the attached pages into the manual according to the collating instructions on the back of this cover. Change bars in the margin indicate technical additions and changes; asterisks denote deletions. There are no new or deleted commands associated with this release. The majority of corrected items relate to Trouble Reports (TRs) and User comments.

Note:

Insert this cover after the manual cover to indicate the updating of the document with Addendum A.

SOFTWARE SUPPORTED

Multics Software Release 10.1

ORDER NUMBER

CC69-00A

November 1982

35837
1182
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

To update the manual, remove old pages and insert new pages as follows:

Remove	Insert
Front Cover	Front Cover
2-3, 2-4	2-3, 2-4
5-9, 5-10	5-9, 5-10
5-15 through 5-18	5-15 through 5-18
5-21, 5-22	5-21, 5-22
5-29 through 5-36	5-29 through 5-36
5-45, 5-46	5-45, 5-46
5-55, 5-56	5-55, 5-56
5-63 through 5-76	5-63 through 5-76
B-1, B-2	B-1, B-2
Remarks Form (CC69-00)	Remarks Form (CC69-00A)

File No.: 1L23

11/82

CC69-00A

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

CUT ALONG LINE

TITLE: MULTICS REPORT PROGRAM GENERATOR (MRPG) REFERENCE MANUAL ADDENDUM A

ORDER NO.: CC69-00A

DATED: NOVEMBER 1982

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME _____ **DATE** _____

TITLE _____

COMPANY _____

ADDRESS _____

_____

PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

FOLD ALONG LINE

FOLD ALONG LINE

CUT ALONG LIN

Honeywell

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario M2J 1W5
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

20646, 1478, Printed in U.S.A.

CC69, Rev. 0