PROGRESS®

LANGUAGE REFERENCE

Printed in U.S.A.

December 1990

Contents

Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Preface

This book is a reference guide to the statements, phrases, functions, operators and special symbols used in the PROGRESS language and in PROGRESS/SQL. Use this book, along with the *PROGRESS Language Tutorial* and the *Programming Handbook*, to write PROGRESS procedures.

In this manual, the term PROGRESS refers to the PROGRESS 4GL/RDBMS (formerly Full PROGRESS). The PROGRESS Application Development System consists of the PROGRESS 4GL/RDBMS and the PROGRESS FAST TRACK Application Builder. For information about PROGRESS FAST TRACK, see the *PROGRESS FAST TRACK Tutorial* and the *PROGRESS FAST TRACK User's Guide.*

THE ORGANIZATION OF THIS BOOK

This book is organized as follows:

Introduction to the PROGRESS Language

Describes the basic components of the PROGRESS language, and presents information about editing, storing, and printing procedures. It also explains how to use expressions in PROGRESS statements.

The PROGRESS Language

Provides detailed information about each PROGRESS statement, phrase, function, operator, and special symbol.

SQL Reserved Words

Provides detailed information about each SQL reserved word used in interactive and embedded SQL in PROGRESS.

TYPOGRAPHICAL CONVENTIONS

This document uses the following typographical conventions:

- **Bold typeface** indicates commands and characters you type. It also emphasizes important points.

- *Italic typeface* indicates a parameter or argument you supply. It also introduces new terms and manual titles.
- `Typewriter typeface` indicates system output and PROGRESS procedures. It also highlights file names, field names, command names, and menu options in text.

LANGUAGE SPECIFICS

This section explains the organization of the reference pages for the PROGRESS language. Special symbols are listed first, followed in alphabetical order by the statements, functions, and operators. The explanation of each language component includes:

- A **Purpose** or description of the language element.
- **Syntax** for the component.
- **Block Properties** for all block statements.
- **Data Movement** diagrams for all data handling statements.
- The **Options** and **arguments** you can use with the statement, phrase, or operator.
- One or more **examples** illustrating the use of the component.
- **Notes** which highlight special cases or provide hints on using the component.
- A **See Also** section listing other related language components.

For SQL keywords only, an additional section is included:

- A **Used In** section indicating whether the SQL reserved word can be used in interactive SQL, embedded SQL, or both.

Syntax

The syntax for each component follows a set of conventions:

- Uppercase words are keywords. Although they are always shown in uppercase, you can use either uppercase or lowercase when using them in a procedure. For example:

SYNTAX

```
ACCUM aggregate-phrase expression
```

Here, ACCUM is a keyword.

- Italics identify options or arguments that you must supply. In the ACCUM function above, *aggregate-phrase* and *expression* are options.

- You must end all statements (except for DO, FOR, and REPEAT) with a period. DO, FOR, and REPEAT statements may end with either a period or a colon. For example:

```
FOR EACH customer:
  DISPLAY name.
END.
```

- Square brackets ([ ]) indicate that an item is optional. For example:

CLEAR [FRAME *frame*] [ALL] [NO-PAUSE]

Here, FRAME *frame*, ALL, and NO-PAUSE are optional. There are some instances where square brackets are not a syntax notation, but part of the language instead. For example, you must use square brackets when referencing array elements. The description tells you when to include the brackets. Otherwise, brackets are part of the syntax notation.

- Braces ({ }) indicate that one of the enclosed items is required. For example:

$$\text{ASSIGN}\begin{Bmatrix} \textit{field} \\ \textit{field} = \textit{expression} \end{Bmatrix} \ \ldots$$

Here, you must select one of *field* or *field = expression*. There are some instances where braces are not a syntax notation, but part of the language instead. For example, a called procedure must use braces when referencing arguments passed by a calling procedure. The description tells you when to include the braces.

- Ellipses (...) indicate that you can choose one or more of the preceding items they follow. If a group of items is enclosed in braces and followed by ellipses, you must choose one or more of those items. If a group of items is enclosed in brackets and followed by ellipses, you can optionally choose one or more of those items. For example:

$$\text{COLOR}\begin{Bmatrix} [\ \text{DISPLAY}\,\textit{color-phrase} \\ \text{PROMPT}\,\textit{color-phrase} \ ] \end{Bmatrix} \ \ldots \ \textit{field} \ \ldots \ [\ \textit{frame- phrase}\ ]$$

Here, you must include one or both of the DISPLAY and PROMPT options. You can also use multiple *field* options.

- A comma followed by ellipses (...) indicate that you can choose one or more of the items they follow. If you choose more than one of a group of items, you must separate those items by commas. If a group of items is enclosed in brackets and is followed by a comma and ellipses, you can optionally choose one or more of those items. If a group of items is enclosed in braces and is followed by a comma and ellipses, you must choose one or more of those items.

Examples

There is usually one or more example procedures for every statement, phrase, function, and operator. Examples use the following conventions:

- They are shown in boxes with borders.

- The name of the procedure is in the upper right corner of the box.

- You can find all procedures in the proguide subdirectory of the directory where you installed the PROGRESS software (by default, \DLC under DOS and OS/2, /usr/dlc under UNIX, [sys]<dlc> under BTOS/CTOS, and [DLC] under VMS). These procedures are stored in a packed format in a single file named refproc in the proguide directory, so that the many small example procedures take up much less disk space. You can unpack all of the reference procedures into individual procedure files or you can extract specific procedures one at a time:

To unpack these*procedures*, access the PROGRESS Procedure Library from the PROGRESS Help menu, as follows:

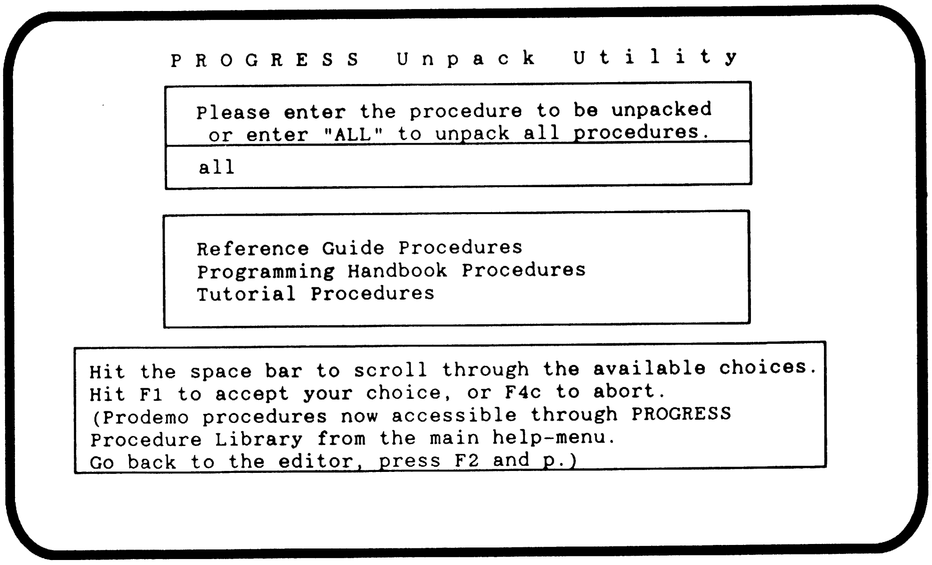
1. Press the HELP (F2) key to display the PROGRESS Help menu.

2. From the Help menu, select option **p**, Access the Procedure Library. The Main Menu of the PROGRESS Procedure Library appears on the screen.

3. From the Library Main Menu, select option **h**, Help, for an overview of the Library contents and instructions on how to use the Library.

From the Main Menu of the PROGRESS Procedure Library, you can also access the procedures used in the *PROGRESS Language Tutorial*, and the *Programming Handbook*, by choosing option **u**, Unpack Utility.

4. When you are prompted for the name of the procedure to unpack, enter **all** and press RETURN. The procedure prompts you to choose the book whose procedures you want to unpack. Choose "Reference Guide Procedures."

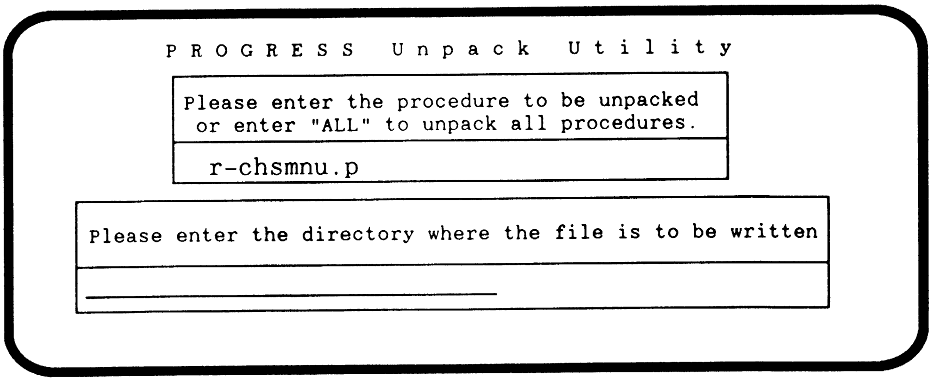


```
PROGRESS Unpack Utility

Please enter the procedure to be unpacked
 or enter "ALL" to unpack all procedures.
all

Reference Guide Procedures
Programming Handbook Procedures
Tutorial Procedures

Hit the space bar to scroll through the available choices.
Hit F1 to accept your choice, or F4c to abort.
(Prodemo procedures now accessible through PROGRESS
Procedure Library from the main help-menu.
Go back to the editor, press F2 and p.)
```

You are then asked to enter the name of the directory where you want the files to be written. If you enter blanks, they will be written to your working directory. If you unpack the procedures into the proguide directory, all users can easily access them.

- To unpack a *specific procedure*, when you are prompted for the name of the procedure to unpack, type the name of the procedure you want to use (e.g. r-chsmnu.p).

To put the procedure in your working directory, enter blanks when prompted for the name of the directory where you wish to place the procedure.



```
PROGRESS Unpack Utility

Please enter the procedure to be unpacked
 or enter "ALL" to unpack all procedures.
 r-chsmnu.p

Please enter the directory where the file is to be written
```

All of the examples are based on the demo database. You make a working copy of this database by using the following commands:

Operating System	To Copy the demo Database
UNIX	`prodb` *database-name* `demo`
DOS and OS/2	`prodb` *database-name* `demo`
VMS	`PROGRESS/CREATE` *database-name* `demo`
BTOS/CTOS	`PROGRESS Create Database` `New Database Name` *database-name* `Copy From Database Name` demo

OTHER USEFUL PUBLICATIONS

The following is a list of publications written by Progress Software Corporation which you may find useful:

PROGRESS Installation Notes

Details step–by–step instructions for installing PROGRESS.

PROGRESS Test Drive

Introduces new users to PROGRESS through a tutorial that describes a sporting goods distributor's inventory and order processing application.

PROGRESS Language Tutorial

Provides a "how–to" guide to PROGRESS fundamentals, designed for both novice and experienced programmers.

PROGRESS Programming Handbook

Details advanced PROGRESS programming techniques.

System Administration I: Environments

Explains the DOS, UNIX, VMS, and BTOS/CTOS concepts required to run PROGRESS, and provides information about running PROGRESS on networks.

System Administration II: General

Describes PROGRESS limits, disk and memory requirements, startup and shutdown procedures, backing up and restoring databases, and PROGRESS utilities. It also provides information about security administration, using multi–volume databases, and Roll Forward Recovery.

Pocket PROGRESS

Lets you quickly look up information about the PROGRESS language or programming environment. There is also a master index for the documentation in this quick reference.

Developer's Toolkit Manual

Explains how to use the PROGRESS Developer's Toolkit, a set of tools used to prepare PROGRESS applications for distribution.

Database Gateways Guide

Provides information about the how to use the PROGRESS 4th generation programming language on different relational database management systems other than PROGRESS RDBMS.

3GL Interface Guide

Supplies information about the PROGRESS Host Language Call (HLC), embedded SQL, and the Host Language Reference (HLI). This manual also contains information on how to use the PROBUILD utility.

Introduction to the PROGRESS Language

You use the PROGRESS language to write **procedures** that make up your PROGRESS application. This book details each of the statements, phrases, functions, operators, and special symbols you can use to write PROGRESS procedures. Use this book, together with the *PROGRESS Language Tutorial* and the *Programming Handbook*, to write applications in PROGRESS.

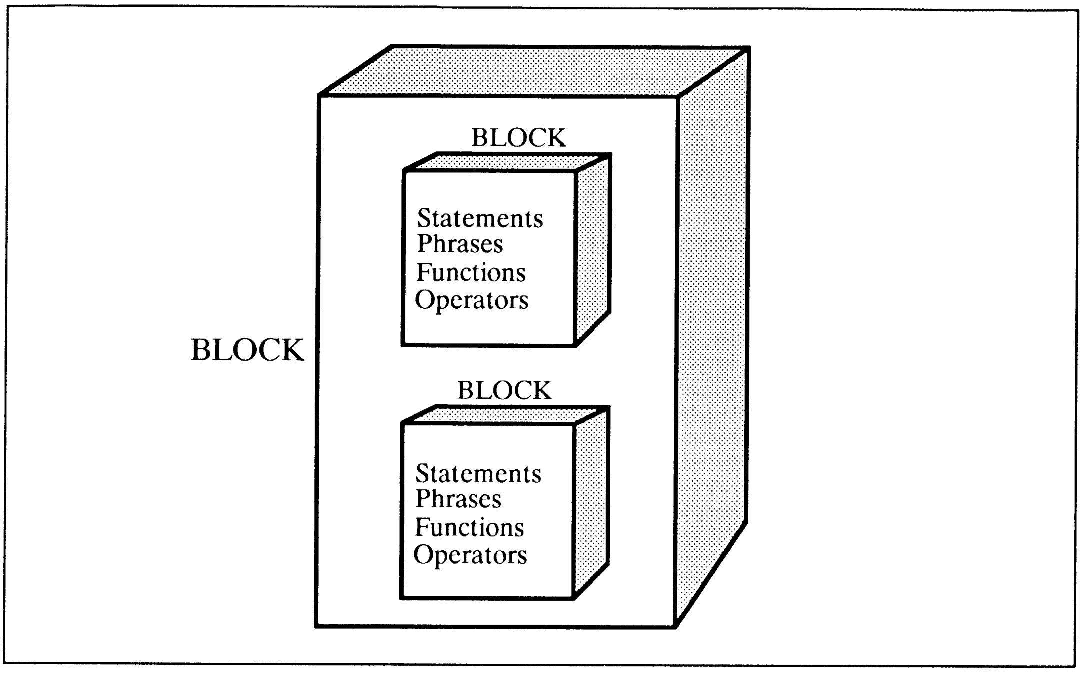
LANGUAGE COMPONENTS

The PROGRESS language consists of:

- **Statements** like DISPLAY, DELETE, and UPDATE that tell PROGRESS what kind of processing to do.
- **Phrases** that modify the way PROGRESS processes statements.
- **Functions** that perform a predefined action like calculating a square root or determining the current date.
- **Operators** such as + (addition), - (subtraction), < (less than), and > (greater than) that let you manipulate different values in a procedure.
- **Special symbols**

LANGUAGE STRUCTURE

PROGRESS is a block structured language. You use blocks to group statements together.



Grouping parts of a procedure into blocks lets you apply processing characteristics to each of those parts. For example, if you want a group of statements to use the same display frame, you can group those statements into a block.

Using blocks also lets you take advantage of the automatic block-level processing services provided by PROGRESS. For more information on these processing services, see Chapter 5 of the *Programming Handbook.*

EDITING PROCEDURES

Procedures are stored in normal ASCII text files. Most of the time you will develop procedures using the PROGRESS editor or any other text editor.

However, you need to be careful if you edit an include file with an editor other than the PROGRESS editor, and then run a procedure which references the include file. In this case, the procedure being run will not necessarily be recompiled automatically. If the procedure being run has not yet been compiled, or has itself been changed since it was last run, then it will be recompiled and the changes you made to the include file will be used. Otherwise, the changes to the include file will not be reflected in the procedure being run and incorrect processing could result. If you use the PROGRESS editor whenever you modify include files, this situation will never be a problem for you. PROGRESS keeps track of the procedures that use each include file and will recompile them if necessary.

STORING PROCEDURES IN FILE SYSTEMS

You can use the hierarchical directory structure of your operating system file system to help you organize your databases and procedures. The procedures you use with one database might be kept in a single directory dedicated to that purpose. For a more complex system with several users who have differing access needs, the database may be kept in a single common directory. Procedures for each category of user can be kept in individual directories. For example, in an accounting system, the procedures for order entry and printing sales slips can be in one directory while the manager's procedures can be in a separate directory.

When you develop and save procedures using the GET (F5) and PUT (F6) keys, the procedures are stored in your working directory by default. If you enter a procedure name for either the GET function or the RUN statement, PROGRESS searches the directories you have defined for the PROPATH variable. In addition to those directories, PROGRESS always searches the directory containing PROGRESS, and the prodemo and proguide subdirectories. You can override this search by giving the complete path name of the procedure you want to use.

If you press GET (F5) to retrieve a procedure, PROGRESS can find the procedure in a directory other than your default directory (on your PROPATH). If you subsequently press PUT (F6) to store the procedure and use the default file name you used to retrieve the procedure, PROGRESS stores the procedure in your working directory. To store the procedure in the directory from which you retrieved it, specify the full path name.

LISTING AND PRINTING PROCEDURES

You can get an alphabetical list of all the files in a directory or print files using your operating system directory command, for example:

Operating System	Directory Command	Print Command
UNIX	`ls`	`lp or lpr`
DOS & OS/2	`dir`	`print`
VMS	`DIRECTORY`	`PRINT`
BTOS/CTOS	`LIST [File List]`	`PRINT [File List]`

These commands display the name of each procedure file because procedures are stored as files.

From within PROGRESS, the DOS statement lets you execute any operating system command. For example, when PROGRESS executes a procedure consisting of the next statement, the names of all files in your working directory are displayed.

Syntax(DOS):

```
DOS dir
```

If you want to use the DOS print command from inside PROGRESS, be sure to use "print" at least once before you start a PROGRESS session. This ensures that "print" is resident in memory, and avoids memory allocation problems while PROGRESS is running.

Syntax(OS/2):

```
OS2 dir
```

From within PROGRESS, the OS2 statement lets you execute any operating system command. For example, when PROGRESS executes a procedure consisting of the next statement, the names of all files in your working directory are displayed.

From within PROGRESS, the UNIX statement lets you execute both "ls" and "lp".

This command prints all the files in the directory /usr/dlc/proguide.

Example(UNIX):

```
UNIX lp /usr/dlc/proguide/*
```

You can get a printout of a procedure by using the operating system print command. For example, the following command prints the procedure "demo1" from your working directory on the printer.

Syntax(VMS):

```
VMS print demo1
```

USING EXPRESSIONS

Expressions are used in many PROGRESS statements and functions. An expression is a constant, field name, variable name, or any combination of these. Table 1 shows some valid expressions.

Table 1: Expressions

Expression	Description
1	a constant
cust-num	a field
test-variable	a variable
integer(max-credit) / 12	the quotient of the integer value of a field

If the value of one or more of the components of an expression is an unknown value (?), the result of the expression is usually an unknown value.

When you use expressions to perform mathematical calculations, note that if the result of operations involving integers overflow the largest value of an integer ($\pm$ 2 billion), PROGRESS may end abnormally.

Evaluating Expressions

When evaluating expressions, PROGRESS uses precedence in determining the result. For example:

```
3 * 5 + 2
```

The result of this expression is different depending on the order in which you process the multiplication and addition operators: 3 * (5 + 2) is 21 but (3 * 5) + 2 is 17.

If an expression contains two operators of equal precedence, PROGRESS evaluates the expression from left to right. If the operators are not of equal precedence, PROGRESS evaluates the operator of higher precedence first. Use parentheses to change the default order used to evaluate an expression.

When using an operator, you must leave a space to the left and to the right of that operator. For example, here is a valid use of an operator:

```
x + y
```

Whereas, this is an invalid use of an operator:

```
x+y
```

Table 2 shows the precedence, or order of processing, of PROGRESS functions and operators.

Table 2: Precedence Of Functions And Operators

Name of Operator	Precedence
- UNARY NEGATIVE + UNARY POSITIVE	7 (highest)
MODULO / DIVISION * MULTIPLICATION	6
- DATE SUBTRACTION - SUBTRACTION + DATE ADDITION + CONCATENATION + ADDITION	5
MATCHES LT or < LE or < = GT or > GE or > = EQ or = NE <> BEGINS	4
NOT	3
AND	2
OR	1 (lowest)

Calculating the Result of an Expression

PROGRESS does calculations using up to 10 decimal places. The value PROGRESS assigns to a database field is rounded based on the Dictionary decimal specification for that field. The value assigned to a variable is rounded based on the DECIMAL specification for the variable, and defaults to 10 decimal places.

: Punctuation

Ends block labels and block header statements like DO, FOR, and REPEAT. Also follows the EDITING keyword in an EDITING phrase and is a device delimiter on VMS.

; Special Character

When combined with a second character in the PROGRESS procedure editor, provides alternative representations of special PROGRESS characters as follows (see also, *Extended Alphabet Support* in Chapter 2 of the *Programming Handbook*).

Special Character	@	[	]	^	‘	{	|	}	~
Alternative Representation	;&	;<	;>	;*	;’	;(	;%	;)	;?

To suppress the semicolon's interpretation as a special character, precede it with a tilde (~). For example, to enter the string ;< in the procedure editor and not have PROGRESS interpret it as an open bracket, type in ~;<.

Finally, if an ASCII character is mapped to an extended alphabetic character by an `IN` statement in the protermcap file, you can enter the extended character in the procedure editor by preceding the ASCII character with a semicolon. For example, if [is mapped to A–umlaut, the ;[sequence is interpreted and stored as A–umlaut.

; Punctuation

In PROGRESS Version 6 and later versions, can be used to terminate statements when -Q is turned on for the session in which the procedure is compiled. Just like a period. This disables the use of the semicolon within, for example, Unix escapes such as “UNIX SMBL=foo; export SMBL”.

. Punctuation

Ends all statements, including block header statements. Also a directory path or file suffix separator in VMS.

, Punctuation

Separates multiple file specifications (used in FOR statements, FOR phrases, and PRESELECT phrases), branching statements (used in UNDO statements and phrases), and multiple arguments of a function.

? Special Character

Represents the unknown value. PROGRESS treats a quoted question mark ("?") in a procedure or in an input file as the question mark character. PROGRESS treats an unquoted question mark (?) in a procedure or in an input file as the unknown value.

When entering data into an input field, if you enter a question mark (?) into the first position of the field and there are no other characters in the field, PROGRESS treats the question mark as the unknown value.

Additional points about the unknown value:

- Any number of unknown value records can be in a unique index. This is useful in cases where you want to defer choosing key values for a unique index.
- If you define a field as mandatory in the Dictionary, that field cannot contain the unknown value when PROGRESS writes the record to the database.
- For sorting and indexing purposes, the unknown value sorts high.
- The question mark (?) character in the first position of a field means "unknown value", not "question mark".
- When using the unknown value in an expression, the result of that expression is usually unknown.

For information about how the unknown value works with logical data types and conditional statements, see the following reference pages: EQ operator, GE operator, GT operator, IF...THEN...ELSE... statement, LE operator, LT operator, NE operator.

\ Special Character

An “escape” character (UNIX only). A directory path separator (DOS and OS/2 only).

~ Special Character

An “escape” character that tells PROGRESS to read the following character literally. A tilde followed by three octal digits represents a single character. Use tilde (~) under DOS, OS/2, BTOS/CTOS, and VMS, or either the tilde (~) or the backslash (\) under UNIX, as a lead-in when entering special characters into the editor. Table 3 lists those special characters. In a procedure, a tilde followed by something other than the items in Table 3 is ignored (for example, “~abc” is treated as “abc”). (This may not work as expected when passing parameters to an include file.) The items in Table 3 are case-sensitive.

Table 3: Entering Special Characters in the Editor

Sequence	Interpreted As	Comment
~"	"	For use within quoted strings as an alternative to two quotes ("")
~'	'	For use within quoted strings as an alternative to two quotes (' ')
~ ~	~	
~\	\	
~{	{	
~nnn	A single octal character	nnn must be between 000 and 377 and all three digits are required
~t	Tab character	Octal 011
~r	Carriage return	Octal 015
~n	New line	Line feed - octal 012
~E	Escape	Octal 033
~b	Backspace	Octal 010
~f	Form feed	Octal 014

" Special Character

Encloses character constants or strings. To use quotes within a quoted character string, you must either use two double quotes (""), which compile to a single double quote ("), or you must put a tilde (~) in front of any quotes within the quoted character string. (This does not work when passing parameters to an include file.)

' Special Character

The single quote functions exactly as the double quote. However, if you use both single and double quotes in a statement, the compiler checks the outermost quotes first, giving them precedence over the innermost quotes. For example, `DISPLAY '"test"'` returns `"test"`. The double quotes are regarded as literals. `DISPLAY "'test2'"` returns `'test2'`.

/ Special Character

A directory path separator (UNIX). Also used for date fields (99/99/99).

( ) Expression Precedence

Raises expression precedence. Also, some functions require you to enclose arguments in parentheses.

[ ] Array Reference

Encloses array subscripts ([1], [2], etc.) or ranges (such as [1 FOR 4]).

In the range, the first element may be a variable and the second must be a constant. The specification [1 FOR 4] tells PROGRESS to start with the first array element to work with that and the next three elements. For more information on using a range of array elements, refer to Chapter 10 in the *PROGRESS Language Tutorial.*

The square brackets are also used as part of a VMS directory specification.

YES, NO, TRUE, FALSE Logical Values

Represent values of logical fields or variables. In a procedure you must use these values even if alternate values are given in the FORMAT specification for a field or variable.

= or EQ Operator

(See EQ Operator)

< or LT Operator

(See LT Operator)

< = or LE Operator

(See LE Operator)

> or GT Operator

(See GT Operator)

> = or GE Operator

(See GE Operator)

< > or NE Operator

(See NE Operator)

{ } Argument Reference

Refers to an argument being passed by a calling procedure, or to an argument in an include file.

PROGRESS converts all arguments to character format. This conversion removes the surrounding quotes if the parameter was specified as a character constant in the RUN or include statement.

When one procedure is RUN (called) from another and arguments are used, PROGRESS recompiles the called procedure, substituting the arguments being passed by the calling procedure, and then runs the called procedure.

SYNTAX

```
{n}
```

```
{ &argument-name }
```

Enter the braces as shown, they do not represent syntax notation here.

n

The number of the argument being referred to. If *n* = 0, PROGRESS substitutes the name of the current procedure (the name you used when you called it, not the full path name) as the argument. If *n* = *, PROGRESS substitutes all arguments being passed by the calling procedure (but not the name {0}). If you refer to the *n*th parameter and the calling procedure does not supply it, {*n*} is ignored.

&argument-name

The name of the argument being referred. If you refer to an *argument-name* and the calling procedure does not supply it, {*&argument-name*} is ignored.

If *argument-name* is an asterisk "*", PROGRESS substitutes all arguments being passed by the calling procedure, adding quotation marks to each parameter, letting you pass the named argument list through multiple levels of include files.

EXAMPLES

r-arg.p

```
RUN r-arg2.p "customer" "name".
```

r-arg2.p

```
➤ FOR EACH {1}:
➤   DISPLAY {2}.
  END.
```

The procedure r-arg.p runs procedure r-arg2.p, passing to r-arg2.p the arguments "customer" and "name". PROGRESS substitutes these arguments for {1} and {2} in the r-arg2.p procedure.

r-inc.p

```
DEFINE VARIABLE txt AS CHARACTER.
DEFINE VARIABLE num AS INTEGER.

txt = "PROGRESS VERSION".
num = 4.

{r-inc.i &int=num &str=txt}
```

r-inc.i

```
MESSAGE {&str}      /* the &str named argument */
        {&int}      /* the &int named argument */
        "Asterisk displays all the arguments:"
        {*}.        /* all the arguments passed
                       by the calling procedure */
```

The procedure r-inc.p defines the variables txt and num, and assigns the values "PROGRESS VERSION" and "4" to them. The r-inc.p procedure calls the r-inc.i file and passes the &int and &str arguments to the include file. Because the parameters are named, ordering them is unimportant. The called procedure will find each argument, regardless of placement. The r-inc.i include file displays a message that consists of the passed arguments. The asterisk argument displays all the parameters as they are listed in the r-inc.p procedure.

NOTES

- If you pass arguments when using the RUN statement, you cannot precompile the called procedure. When PROGRESS compiles a procedure, it must have all the values the procedure needs. So, if you pass arguments to a procedure you are calling with the RUN statement, PROGRESS evaluates those arguments at run time, not at compile time.
- You can use the name of an include file as an argument to another include file. For example, a reference to {{1}} in an included procedure causes the statements from the file whose name is passed as the first argument to be included.
- PROGRESS disregards an empty pair of braces ({}).
- The maximum length of the parameters you can pass to an include file is 2K.

SEE ALSO { } Include Files, COMPILE Statement, RUN Statement, ; Special Character

{ } Include File

If PROGRESS encounters the name of a file enclosed in braces ({ }) when compiling a procedure, it retrieves the statements in that file and compiles them as part of the main procedure. You can name arguments you want substituted in the file before compilation.

SYNTAX

$$\left\{ \textit{include-file} \begin{bmatrix} \textit{argument} & \\ \&\textit{argument-name} & = \text{“}\textit{argument-value}\text{”} \end{bmatrix} \ldots \right\}$$

Enter the braces ({ }) as shown, they do not represent syntax here.

include-file

The name of an external operating system file which contains statements to be included during compilation of a procedure. This filename follows normal DOS, OS/2, UNIX, BTOS/CTOS, and VMS naming conventions and is case sensitive under UNIX. If the file you name has an unqualified path name, PROGRESS searches directories based on the PROPATH environment variable.

argument

A value to be used by *include-file*. When PROGRESS compiles the main procedure (the procedure containing the braces), it copies the contents of *include-file* into that procedure, substituting any *arguments*. The first argument replaces {1} in the included file, the second argument replaces {2}, and so on. Therefore, you can use included procedures with arguments even when you are precompiling a procedure.

&argument-name = "argument-value"

argument-name is the name of the argument you want to pass to the include file. You can use variable names, field names, and reserved words as argument names.

argument-value is the value of the argument you are passing to the include file. You must enclose the *argument-value* in quotation marks.

EXAMPLES

```
                                              r-incl.p
    FOR EACH customer:
➤     {r-fcust.i}
➤     {r-dcust.i}
    END.
```

r-fcust.i

```
FORM customer.cust-num customer.name LABEL "Customer Name"
     customer.phone FORMAT "999-999-9999".
```

r-dcust.i

```
DISPLAY customer.cust-num customer.name customer.phone.
```

The main procedure uses externally defined and maintained files for the layout and display of a customer report. You can use these same include files in many procedures.

r-incl2.p

```
    DEFINE VARIABLE var1 AS INTEGER.
    DEFINE VARIABLE var2 AS DECIMAL.
    DEFINE VARIABLE var3 AS LOGICAL.
                  .
                  . /* any statements */
                  .
➤   {r-show.i point-A var1 var2 var3}
                  .
                  . /* any statements */
                  .
```

r-show.i

```
/* show include file */
MESSAGE "At" "{1}" "{1}" "{2}" "{3}" "{3}" "{4}" "{4}".
```

When the main procedure is compiled, the line referencing the show include file is replaced by the line:

```
MESSAGE "At" "point-A" "var1" "var1" "var2" "var2"
"var3" "var3" "var4" "var4".
```

This example shows how you can use include files to extend the PROGRESS language. The main procedure uses a new statement, r-show.i, to display the values of fields or variables at various points in a procedure. The include file in this example can handle up to 4 passed arguments. The main procedure only passes 3 (point-A, var1, var2, and var3).

r-custin.p

```
FOR EACH customer:
  {r-cstord.i &frame-options = "CENTERED ROW 3 NO-LABEL"}.
  UPDATE cust.cust-num name address address2 city st zip
         phone max-credit WITH FRAME cust-ord.
END.
```

r-cstord.i

```
FORM "Cust #" AT 1 customer.cust-num AT 10 SKIP(1)
     customer.name AT 10
     customer.address AT 10
     customer.address2 AT 10
     customer.city AT 10 customer.city customer.st
                   customer.zip SKIP(1)
     "Phone" AT 1 customer.phone
     FORMAT "999/999-9999" AT 10
     "Max Crd" AT 1 customer.max-credit AT 10
     WITH FRAME cust-ord OVERLAY {&frame-options}.
```

The r-custin.p procedure displays, for each customer, a frame in which you can update some customer information. The procedure calls the r-cstord.i file and passes the named argument &frame-options, and the value of the argument (CENTERED ROW 3 NO-LABEL) to the include file. When the include file references the &frame-options argument, it uses the value of the argument, and therefore displays the OVERLAY frame cust-ord as a centered frame at row 3 without a label.

NOTES

- When you use braces to include one procedure in another procedure, PROGRESS includes the second procedure when it compiles the first procedure. This method has the same effect as if you were to use the editor to copy the statements into the main procedure, but separate include files are often easier to maintain.

- Using include files with parameters is especially useful when you have a base procedure and want to make several copies of it, changing it slightly each time. For example, you might want to change only the name of some files or fields used by the procedure.

- Instead of maintaining duplicate source files, create a single include file with the variable portions (such as the names of files and fields) replaced by {1}, {2}, and so on. Then each procedure you write can use that include file, passing file and field names as arguments.

- Include files can be nested. That is, an include file can contain references to other include files.

- You can use the name of an include file as an argument to another include file. For example, a reference to {{1}} in an include file causes the statements from the file whose name is passed as the first argument to be included.

- PROGRESS disregards an empty pair of braces ({}).

- Include files are particularly useful for using form layouts in multiple procedures, especially if you do not include the keyword FORM or the closing period (.) of the FORM statement. For example:

r-cust.f

```
customer.cust-num
customer.name
SKIP (2)
customer.st
```

r-incl3.p

```
FORM {r-cust.f}.
```

r-incl4.p

```
FOR EACH customer:
  DISPLAY {r-cust.f} WITH 3 DOWN.
END.
```

- The r-incl3.p procedure includes the r-cust.f file as the definition of a FORM statement, while the r-incl4.p procedure uses the include file as a layout for a DISPLAY statement.

- If you use double quotes (" ") around arguments in an argument list, then they will be stripped off. If you use single quotes (' '), they WILL be passed. If you WANT to pass one set of double quotes, you have to use four sets of double quotes.

SEE ALSO { } Argument Reference, COMPILE Statement, RUN Statement

/* Comments */

Lets you add explanatory text to a procedure between the /* and */ characters.

SYNTAX

```
/* comment */
```

comment

Descriptive text.

EXAMPLES

In the following example, comments are used to document the history of modifications made to the procedure:

```
r-comm.p

/* Procedure written 9/5/87 by CH; revised 9/27/87 by DG */
FOR EACH customer:
  DISPLAY cust-num name contact phone.
END.
```

In the next example, comments are used to describe what the procedure is doing:

```
r-comm2.p

/* step through unshipped orders */
FOR EACH order WHERE shipped = "":
  DISPLAY odate pdate terms.
  /* display order date, promise date, terms */
/*
  FOR EACH order-line OF order:
    /* display all order-lines of each order */
    DISPLAY order-line.
  END.
*/
END.
```

The comment symbols enclosing the inner FOR EACH block turn that block into a comment for testing purposes. Because you can nest comments, any comments already in the code being bypassed are processed correctly.

NOTES

- You can nest comments. That is, you can place a comment inside another comment.

+ Unary Positive Operator

Preserves the positive or negative value of a numeric expression. Be careful not to confuse this operator with the addition operator which you use to add expressions together.

SYNTAX

```
+ expression
```

expression
A constant, field name, variable name, or any combination of these whose value is numeric.

EXAMPLE

r-unpos.p

```
DEFINE VARIABLE old-max LIKE max-credit LABEL "Old Max Cr".
FOR EACH customer:
  old-max = + max-credit.
  max-credit = + (max-credit + 100).
  DISPLAY name old-max max-credit.
END.
```

In this example, the sign of max-credit is preserved as is the sign of the sum of max-credit + 100. Here, the unary positive is not really necessary and is used only to document the procedure.

+ Addition Operator

Adds two numeric expressions.

SYNTAX

```
expression   + expression
```

expression

A constant, field name, variable name, or any combination of these whose value is numeric.

EXAMPLE

r-addn.p

```
    FOR EACH customer:
-->   max-credit = max-credit + 100.
    END.
```

The addition operator (+) adds 100 to the value of the max-credit field.

NOTE

- The addition of two decimal expressions produces a decimal result; the addition of two integer expressions produces an integer result; the sum of an integer expression and a decimal expression is a decimal expression.

+ Concatenation Operator

Produces a character value by joining, or concatenating, two character strings or expressions.

SYNTAX

expression + *expression*

expression

A constant, field name, variable name, or any combination of these whose value is a character string.

EXAMPLE

r-conc.p

```
FOR EACH customer:
  DISPLAY SKIP (1) name SKIP address SKIP
    city + ", " + st FORMAT "x(16)" zip SKIP(2).
END.
```

This procedure prints mailing labels. It uses the concatenation operator (+) to ensure that the third line of each label shows the city and state separated by a comma and a space. The FORMAT x(16) is specified to provide room for up to 16 characters in the result of the concatenation. If a FORMAT was not given then only the first 8 characters of the result would be displayed because x(8) is the default format for a character expression.

Here is a label produced by this procedure:

```
Second Skin Scuba
79 Farrar Ave
Yuma, AZ         85369
```

+ Date Addition Operator

Adds a number of days to a date, producing a date result.

SYNTAX

```
date + days
```

date

An expression (a constant, field name, variable name, or any combination of these) whose value is a date.

days

An expression whose value is the number of days you want to add to a *date*.

EXAMPLE

r-dadd.p

```
DISPLAY "ORDERS SCHEDULED TO SHIP MORE THAN ONE WEEK LATE".
FOR EACH order WHERE shipped = "":
➤ IF sdate > (pdate + 7)
  THEN DO:
    FIND customer OF order.
    DISPLAY order.order-num order.cust-num
            customer.name pdate sdate customer.terms.
  END.
END.
```

This procedure finds all orders that have not been shipped. If the ship date is more than 1 week later than the promise date, the procedure finds the record for the customer who placed the order and displays order and customer data.

NOTE

- The date addition operator rounds days to the nearest integer value.

- Unary Negative Operator

Reverses the sign of a numeric expression. Be careful not to confuse this operator with the subtraction operator which subtracts one expression from another.

SYNTAX

```
- expression
```

expression
A constant, field name, variable name, or any combination of these whose value is numeric.

EXAMPLE

r-uneg.p

```
DEFINE VARIABLE x AS DECIMAL LABEL "X".
DEFINE VARIABLE abs-x AS DECIMAL LABEL "ABS(X)".

REPEAT:
  SET x.
  IF x < 0
  THEN abs-x =- x
  ELSE abs-x = x.
  DISPLAY abs-x.
END.
```

Here, if the user supplies a negative value for the variable x, the procedure uses the unary negative operator (-) to reverse the sign of x, producing the absolute value of x (abs-x).

- Subtraction Operator

Subtracts one numeric expression from another numeric expression.

SYNTAX

```
expression  - expression
```

expression

A constant, field name, variable name, or any combination of these whose value is numeric.

EXAMPLE

```
r-subt.p

DEFINE VARIABLE free-stock LIKE on-hand LABEL "Free Stock".
FOR EACH item:
➤free-stock = on-hand - alloc.
  DISPLAY item-num idesc on-hand alloc free-stock.
END.
```

This procedure determines the amount of inventory available by subtracting the amount allocated from the total on-hand.

NOTE

- The subtraction of two decimal expressions produces a decimal result; the subtraction of two integer expressions produces an integer result. The subtraction of an integer expression from a decimal expression or vice versa produces a decimal result.

- Date Subtraction Operator

Subtracts a number of days from a date to produce a date result, or subtracts one date from another to produce an integer result representing the number of days between the two dates.

SYNTAX

$$\textit{date} - \left\{ \begin{array}{l} \textit{days} \\ \textit{date} \end{array} \right\}$$

date

An expression (a constant, field name, variable name, or any combination of these) whose value is a date.

days

An expression whose value is the number of days you want to subtract from a *date*.

EXAMPLE

r-dsub.p

```
DISPLAY "ORDERS SCHEDULED TO SHIP MORE THAN ONE WEEK LATE".
FOR EACH order WHERE shipped eq "":
  IF (sdate - 7) > pdate
  THEN DISPLAY order.order-num order.cust-num pdate sdate
               (sdate - pdate) LABEL "Days Late".
END.
```

This procedure finds all orders that have not been shipped. If the ship date is more than 1 week later than the promise date, the procedure finds the customers who placed the orders, and displays order and customer data.

NOTE

- The date subtraction operator rounds days to the nearest integer value.

* Multiplication Operator

Multiplies two numeric expressions.

SYNTAX

```
expression * expression
```

expression

A constant, field name, variable name, or any combination of these whose value is numeric.

EXAMPLE

r-mult.p

```
DEFINE VARIABLE inv-value AS DECIMAL LABEL "VALUE".
FOR EACH item:
  inv-value = on-hand * cost.
  IF inv-value < 0
  THEN inv-value = 0.
  DISPLAY item.item-num idesc on-hand cost inv-value.
END.
```

This procedure computes the value of the on-hand inventory for each item. If the on-hand inventory is negative, the inventory value is set to zero.

NOTE

- The product of two decimal expressions is a decimal value; the product of two integer expressions is an integer value; the product of an integer expression and a decimal expression is a decimal value.

/ Division Operator

Divides one numeric expression by another numeric expression, producing a decimal result.

SYNTAX

```
expression / expression
```

expression

A constant, field name, variable name, or any combination of these that results in a numeric value.

EXAMPLE

r-div.p

```
DISPLAY "INVENTORY COMMITMENTS AS PERCENT OF UNITS ON HAND".

FOR EACH item:
  DISPLAY item.item-num idesc alloc on-hand
     ➤ (alloc / on-hand ) * 100 FORMAT ">>9" LABEL "PCT".
END.
```

This procedure divides the number of items allocated by the number of items on hand, producing a decimal value. The multiplication operator (*) converts that decimal value to a percentage.

NOTES

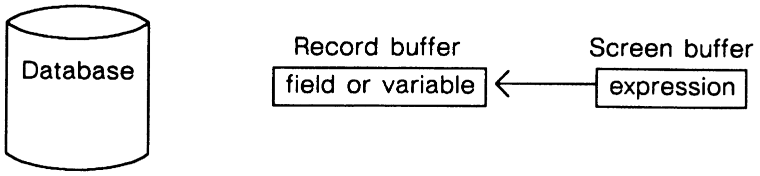
- PROGRESS **always** performs division as a decimal operation (the result of 5 / 2 is 2.5, **not** 2). If you assign the result to an integer field, PROGRESS rounds the decimal to make the assignment. When you want a quotient to be truncated to an integer, use the TRUNCATE function (TRUNCATE(5 / 2, 0) is 2).

- The result of dividing a number by 0 is the unknown value (?). PROGRESS does not display an error message.

= Assignment Statement

Assigns the value of an expression to a database field or variable.

DATA MOVEMENT



SYNTAX

```
field = expression
```

field

The name of a database field or variable to which you want to assign the value of the expression. If the field is an array, and you do not name a particular element, PROGRESS stores *expression* in each element of the array. If you name a particular element, PROGRESS stores *expression* in that element.

The left-hand side of an assignment can also be FRAME-VALUE (see the FRAME-VALUE Statement), SUBSTRING (see the SUBSTRING Statement), or OVERLAY (see the OVERLAY Statement).

expression

A constant, field name, variable name, or any combination of these whose data type is consistent with that of the *field*. If *field* is integer and *expression* is decimal, then the value of the expression is rounded before being assigned. If *field* is decimal and *expression* is decimal, then the value of the expression is rounded to the number of decimal places defined for the field in the Dictionary or defined or implied for a variable.

EXAMPLE

r-asgmnt.p

```
DEFINE VARIABLE ctr AS INTEGER.

FOR EACH customer:
  DO ctr= 1 TO 120:
          customer.mnth-sales[ctr] = 0.
  END.
  ytd-sls = 0.
END.
```

This procedure resets all the monthly sales values and the year-to-date sales to 0 in all customer records. If you want to set values for individual array elements, you can do so by making an explicit assignment using the assignment statement and a specific array reference, such as a[1] or a[i].

NOTE

- You can embed an assignment in a SET or UPDATE statement.

ACCUM Function

Returns the value of an aggregate expression that has been calculated by an ACCUMULATE or DISPLAY statement.

SYNTAX

```
ACCUM  aggregate-phrase expression
```

aggregate-phrase

Identifies the aggregate value to be returned. Here is the syntax for aggregate-phrase:

```
 { AVERAGE       }
 { COUNT         }
 { MAXIMUM       }
 { MINIMUM       }
 { TOTAL         }
 { SUB-AVERAGE   }  ...  [ BY break-group  ] ...
 { SUB-COUNT     }
 { SUB-MAXIMUM   }
 { SUB-MINIMUM   }
 { SUB-TOTAL     }
```

For more information, see the Aggregate Phrase reference page.

expression

An expression (constant, field name, variable name, or any combination of these) that was used in an earlier ACCUMULATE or DISPLAY statement. The expression you use in the ACCUMULATE or DISPLAY statement and the expression you use in the ACCUM function must be in exactly the same form (e.g. on-hand * cost and cost * on-hand are not exactly the same form). For the AVERAGE, SUB-AVERAGE, TOTAL, and SUB-TOTAL aggregate-phrases, *expression* must be numeric.

EXAMPLE

r-accum.p

```
FOR EACH order:
  DISPLAY order-num name odate pdate sdate.
  FOR EACH order-line OF order:
    DISPLAY line-num item-num qty price.
    DISPLAY qty * price LABEL "Ext Price".
    ACCUMULATE qty * price (TOTAL).
    DISPLAY (ACCUM TOTAL qty * price) LABEL "Accum Total".
  END.
  DISPLAY (ACCUM TOTAL qty * price) LABEL "Total".
END.
DISPLAY (ACCUM TOTAL qty * price)
        LABEL "Grand Total" WITH ROW 1.
```

Here, the extended price of each item on an order is totaled. The running total of the order is displayed as well as the order total and grand total for all orders. Totals are accumulated at three levels in this procedure.

SEE ALSO ACCUMULATE Statement, DISPLAY Statement

ACCUMULATE Statement

Calculates one or more aggregate values of an expression during the iterations of a block. Use the ACCUM function to access the result of this accumulation.

SYNTAX

```
ACCUMULATE { expression (aggregate-phrase) } ...
```

expression

A constant, field, variable, or any combination of these whose aggregate value you want to calculate. The expression you use in the ACCUMULATE statement and the expression you use in the ACCUM function (when using the result of the ACCUMULATE statement) must be in exactly the same form. For example, A * B and B * A are not in the same form.

aggregate-phrase

Identifies one or more values to be calculated based on a change in expression or break group. Here is the syntax for aggregate-phrase:

```
{ AVERAGE       }
{ COUNT         }
{ MAXIMUM       }
{ MINIMUM       }
{ TOTAL         }
{ SUB-AVERAGE   }  ...  [ BY break-group ] ...
{ SUB-COUNT     }
{ SUB-MAXIMUM   }
{ SUB-MINIMUM   }
{ SUB-TOTAL     }
```

For more information, see the Aggregate Phrase reference page.

EXAMPLES

r-acmlt.p

```
 FOR EACH customer:
➤ ACCUMULATE max-credit (AVERAGE COUNT MAXIMUM).
END.
DISPLAY "MAX CREDIT STATISTICS FOR ALL CUSTOMERS:" SKIP(2)
        "AVERAGE =" (ACCUM AVERAGE max-credit) SKIP(1)
        "MAXIMUM =" (ACCUM MAXIMUM max-credit) SKIP(1)
        "NUMBER OF CUSTOMERS ="
        (ACCUM COUNT max-credit) SKIP(1) WITH NO-LABELS.
```

This procedure calculates and displays statistics for all customers. It does not show the detail for each customer.

r-acmlt2.p

```
 FOR EACH item:
➤ ACCUMULATE on-hand * cost (TOTAL).
END.
FOR EACH item BY on-hand * cost DESCENDING:
  DISPLAY item-num on-hand cost on-hand * cost
          LABEL "Value" 100 * (on-hand * cost) /
          (ACCUM TOTAL on-hand * cost) LABEL "Value %".
END.
```

This procedure lists each item with its inventory value and lists that value as a percentage of the total inventory value of all items. Items are sorted starting with those of the highest value.

r-acc.p

```
FOR EACH customer BREAK BY sales-rep BY st:
  ACCUMULATE ytd-sls (TOTAL BY sales-rep BY st).
  DISPLAY sales-rep WHEN FIRST-OF(sales-rep) st name ytd-sls.
  IF LAST-OF(st)
  THEN DISPLAY ACCUM TOTAL BY st ytd-sales
       COLUMN-LABEL "State!Total".
  IF LAST-OF(sales-rep)
  THEN DO:
    DISPLAY sales-rep ACCUM TOTAL BY sales-rep ytd-sls
            COLUMN-LABEL "Sales-Rep!Total".
    DOWN 1.
  END.
END.
```

This procedure displays all customers, sorted by sales rep, and by state within the list for each sales rep. The procedure calculates year-to-date sales for each customer, year-to-date sales for each state, and year-to-date sales totals for each sales rep.

NOTE

- The ACCUMULATE statement can only be used in blocks with the implicit looping property. PROGRESS automatically supplies looping services to REPEAT and FOR EACH blocks.

SEE ALSO ACCUM Function, Aggregate Phrase, DISPLAY Statement

Aggregate Phrase

Identifies one or more values to be calculated based on a change in an expression or break group.

SYNTAX

$$\left\{\begin{array}{l}\text{AVERAGE}\\ \text{COUNT}\\ \text{MAXIMUM}\\ \text{MINIMUM}\\ \text{TOTAL}\\ \text{SUB-AVERAGE}\\ \text{SUB-COUNT}\\ \text{SUB-MAXIMUM}\\ \text{SUB-MINIMUM}\\ \text{SUB-TOTAL}\end{array}\right\} \cdots \ [\ \text{BY } \textit{break-group} \ ] \cdots$$

AVERAGE
: Calculates the average of all of the values of the expression in a break group and the average of all of the values of the expression.

COUNT
: Calculates the number of times the expression was counted in a break group and the count of all the values.

MAXIMUM
: Calculates the maximum of all of the values of the expression in a break group and the maximum of all the values of the expression.

MINIMUM
: Calculates the minimum of all of the values of the expression in a break group and the minimum of all the values of the expression.

TOTAL
: Calculates the subtotal of all of the values of the expression in a break group and the grand total of all of the values of the expression.

SUB-AVERAGE
: The average of the values in a break group. Does not supply an average for all records, just those in each break group.

SUB-COUNT
: The number of times an expression has been counted in a break group. Does not supply a count for all records, just those in each break group.

SUB-MAXIMUM
: The maximum value of an expression in a break group. Does not supply a maximum value for all records, just those in each break group.

SUB-MINIMUM
: The minimum value of an expression in a break group. Does not supply a minimum value for all records, just those in each break group.

SUB-TOTAL
: The subtotal of all of the values of the expression in a break group. Does not supply a total value for all records, just those in each break group.

BY *break-group*
: If you use the BREAK option in a FOR EACH block header, you can perform aggregation for groups you name with *break-group* in the BY *break-group* option. ACCUMULATE performs the aggregate calculation for each break group.

EXAMPLE

r-aggreg.p

```
FOR EACH customer BREAK BY st:
  DISPLAY name st ytd-sls (SUB-TOTAL BY st).
END.
```

This procedure lists the customer information for all customers (categorized by state) and a subtotal of each state's year-to-date sales. A grand total would be displayed if the TOTAL aggregate had been used in place of SUB-TOTAL.

NOTES

- In the following procedure, PROGRESS displays the result of the COUNT aggregate even though no accumulation has occurred:

```
DEFINE VARIABLE prntr AS LOGICAL INITIAL FALSE.
FOR EACH item:
  IF prntr
  THEN DISPLAY idesc cost(COUNT) WITH FRAME pr.
END.
```

- By default, PROGRESS displays the aggregate result when the aggregate group ends, as long as the block iterates. If you want to suppress automatic display of zero aggregates, use ACCUMULATE to perform the calculation and test the result with ACCUM before doing the display.

- When you use aggregate phrases to accumulate values within shared frames, you must include the ACCUM option in the Frame Phrase. See the Frame Phrase for more information.

SEE ALSO ACCUMULATE Statement, FOR EACH Statement

ALIAS Function

The ALIAS function returns the alias corresponding to the integer value of expression.

SYNTAX

```
ALIAS (integer-expression)
```

integer-expression

If there are, for example, three currently defined aliases, then `ALIAS(1)`, `ALIAS(2)`, and `ALIAS(3)` return them. Also, continuing the same example of three defined aliases, `ALIAS(4)`, `ALIAS(5)`, etc., return the ? value.

EXAMPLE

r-aliasf.p

```
DEF VAR i AS INT.
REPEAT i = 1 TO NUM-ALIASES.
      DISPLAY ALIAS(i) LABEL "Alias"
             LDBNAME(ALIAS(i)) LABEL "Logical Database".
END.
```

This procedure displays the aliases and logical names of all connected databases.

SEE ALSO CONNECT, DISCONNECT, CREATE ALIAS, and DELETE ALIAS statements; CONNECTED, PDBNAME, SDBNAME, DBTYPE, DBRESTRICTIONS, GATEWAYS, FRAME-DB, NUM-ALIASES, SDBNAME, and NUM-DBS functions.

ALTER TABLE Statement (SQL)

Adds new columns to a table, deletes columns from a table, or changes the format or labels associated with an existing column.

SYNTAX

```
ALTER TABLE table-name  { ADD COLUMN column-name datatype
                              [ FORMAT string ] [ LABEL string ]
                            [ COLUMN-LABEL string [! string] ...]
                            [ [ NOT] CASE-SENSITIVE ]
                              [ DEFAULT initial-value ]
                            DROP COLUMN column-name
                            ALTER COLUMN column-name
                              [ FORMAT string ] [ LABEL string ]
                            [ COLUMN-LABEL string [! string] ...]
                            [ [ NOT] CASE-SENSITIVE ]
                              [ DEFAULT initial-value ]          }
```

table-name
The name of the table you want to change.

ADD COLUMN
Adds a new column to the specified table.

column-name
The name of the new column you want to add to the table.

datatype
The data type of the new column you want to add. The PROGRESS/SQL data types are: CHARACTER, INTEGER, SMALLINT, DECIMAL, FLOAT, DATE, REAL, NUMERIC, and LOGICAL. Refer to Chapter 4 for additional information about data types.

FORMAT *string*
Specifies the display format for the column. The *string* must be enclosed in either single or double quotation marks. See Chapter 4 for information about the various types of display formats.

LABEL *string*
Specifies a label for the column. The *string* must be enclosed in either single or double quotation marks. See Chapter 7 for information about labels.

COLUMN-LABEL *string*
: Specifies a label for the column when its values are displayed vertically (in columns) on the screen or in a printed report. The *string* must be enclosed in either single or double quotation marks. See Chapter 7 for information about column labels.

[NOT] CASE-SENSITIVE
: Indicates whether the values in a character column and comparisons made to it are to be case-sensitive. The default is case-sensitive if you used the ANSI SQL (-Q) startup option. Otherwise, the default is not case-sensitive. The [NOT] CASE-SENSITIVE option cannot be specified on ALTER COLUMN if the column being altered already participates in any indexes.

DEFAULT *initial-value*
: Assigns a default value for the column. This is the same as setting the default value for a field in the PROGRESS Data Dictionary.

DROP COLUMN *column-name*
: Deletes the specified column from the table.

ALTER COLUMN *column-name*
: Changes the display format, label, default value, and/or the column label of the specified column.

EXAMPLES

```
ALTER TABLE cust_table
    ADD COLUMN contact CHARACTER (30).
```

```
ALTER TABLE cust_table
    DROP COLUMN sales_rep.
```

```
ALTER TABLE cust_table
        ALTER COLUMN contact
        LABEL 'Customer Contact'.
```

NOTES

- Only the owner of the table can use the ALTER TABLE statement.
- When adding a new column to a table, you cannot use the NOT NULL and UNIQUE clauses, unless the DEFAULT statement is used. If it is not used, PROGRESS/SQL inserts null values into all existing rows for the new column.
- The update privilege to a new column added through the ALTER TABLE statement is set to the owner of the table. You have to grant update privilege to other users explicitly.
- If you use the ANSI SQL (-Q) option to enforce strict ANSI SQL conformance, all columns added by the ALTER TABLE statement are case-sensitive by default. Use the NOT CASE-SENSITIVE option for each column for which you want to override the default. See Chapter 3 of the *System Administration II: General* book for more information about the -Q startup option.
- You should use the CASE-SENSITIVE option only when it is important to distinguish between uppercase and lowercase values entered for a character column. For example, you would need to use CASE-SENSITIVE to define a column for a part number that contains mixed uppercase and lowercase characters.

AMBIGUOUS Function

Returns a TRUE value if the last FIND statement for a particular record found more than one record that met the index criteria specified.

SYNTAX

```
AMBIGUOUS record
```

record

The name of a record or record buffer used in an earlier FIND statement.

To access a record in a file defined for multiple databases, you may need to qualify the record's filename with the database name. See the description of the Record Phrase for more information.

EXAMPLE

r-ambig.p

```
DEFINE VARIABLE cname LIKE customer.name LABEL "Cust Name".
REPEAT:
  SET cname.
  FIND customer WHERE name = cname NO-ERROR.
  IF AVAILABLE customer
  THEN DISPLAY cust-num address city st zip.
➤ ELSE IF AMBIGUOUS customer
   THEN MESSAGE "There is more than one
               customer with that name".
  ELSE MESSAGE "There is no customer with  that name".
END.
```

This procedure retrieves a customer record based on a name (cname) supplied by the user. If the procedure finds a record, it displays fields from that record. If it does not find a record because more than one record matched the selection criteria (name = cname), it displays the message "There is more than one customer with that name." If it does not find a record because no records matched the selection criteria, it displays a different message.

NOTES

- Sometimes the AMBIGUOUS function returns a TRUE value when there is no ambiguity. See the FIND statement for details.
- AMBIGUOUS is useful only when there is an index. Therefore, if you use the AMBIGUOUS function to test a work file record, the function returns a value of FALSE, because work files do not have indexes.

SEE ALSO AVAILABLE Function, FIND Statement, LOCKED Function, NEW Function

AND Operator

Returns a TRUE value if each of two logical expressions is true.

SYNTAX

expression AND *expression*

expression
A constant, field name, variable name, or any combination of these whose value is logical (true or false).

EXAMPLE

r-and.p

```
DEFINE VARIABLE low-credit LIKE max-credit
  LABEL "Low Credit Limit".
DEFINE VARIABLE hi-credit LIKE max-credit
  LABEL "High Credit Limit".

REPEAT:
  SET low-credit hi-credit WITH FRAME cr-range.
  FOR EACH customer WHERE
    (max-credit >= low-credit) AND
    (max-credit <= hi-credit):
       DISPLAY cust-num name max-credit.
  END.
END.
```

This procedure lists all customers with credit limits between two values (supplied by the user and stored in the variables low-credit and hi-credit). The expressions "max-credit > = low-credit" and "max-credit < = hi-credit" are logical expressions because they yields either a true or false value. Using the AND function to join these logical expressions results in a logical expression that follows the WHERE keyword.

SEE ALSO NOT Function, OR Function

APPLY Statement

In an EDITING Phrase, performs the function of a specified keyboard key. Suppose that when a user presses a key, you test the value of that key to determine whether or not to take some action. You do not want the key to perform its function until you finish testing. When you determine what action to take, you can use the APPLY statement within an EDITING Phrase to perform the function of the key.

Outside an EDITING Phrase, the expressions you APPLY can be any key code whose function is one of HELP, END-ERROR, ERROR, ENDKEY, or STOP.

SYNTAX

```
APPLY expression
```

expression

A constant, field name, variable name, or any combination of these whose value is the keycode of the key whose function you want to apply. A special value of *expression* is the value of the LASTKEY function (this value is always the key code of the last key read from the keyboard or the last character read from an input file). See Chapter 6 of the *Programming Handbook* for a discussion on monitoring and controlling data entry. See Chapter 2 of the *Programming Handbook* for a list of key codes.

EXAMPLE

The r-apply.p procedure creates new items and uses an "answer wheel" for entry of the prod-line field. Whenever the cursor is on the prod-line field, pressing the space bar cycles through the allowed values (equip, unifrm, other). The EDITING Phrase within the UPDATE statement processes keystrokes that occur in the prod-line field and uses the APPLY statement to take normal action in all other fields being updated.

r-apply.p

```
DEFINE VARIABLE prod-list LIKE item.prod-line EXTENT 3.
DEFINE VARIABLE i AS INTEGER.

prod-list[1] = "equip".
prod-list[2] = "unifrm".
prod-list[3] = "other".

REPEAT:
  CREATE item.
  i = 1.
  prod-line = prod-list[i].
  UPDATE item-num idesc prod-line HELP "Press space bar to
         cycle through allowed values"
         /* Edit each keystroke during UPDATE */
    EDITING:
      READKEY. /* If the cursor is in prod-line field
                  then test the keystroke */
      IF FRAME-FIELD = "prod-line"
      THEN DO: /* If the user pressed the space bar
                  increment i, displaying the next
                  allowed value for prod-line */
        IF LASTKEY = KEYCODE(" ")
        THEN DO:   /* On the first time through, i = 1,
                      so i MODULO 3 is the remainder of
                      1 divided by 3 thus i = i+ 1.  */
          i = (i MODULO 3) + 1.
          prod-line = prod-list[i].
          DISPLAY prod-line.
          NEXT.  /* If the user presses anything except
                    RETURN,TAB,BACK TAB,GO,HOME,HELP, or
                    END-ERROR, ring the bell and redo
                    the EDITING-Phrase */
        END.
        IF LOOKUP(KEYFUNCTION(LASTKEY),"return,tab,back-tab,
           go, home, help, end-error") = 0
        THEN DO:
          BELL.
          NEXT.
        END.
      END.    /* If the cursor is not in prod-line field
                 APPLY the keystroke without testing  */
 ➤   APPLY LASTKEY.
    END. /* EDITING-Phrase */
  END.
```

NOTES

- If a procedure calls another procedure from within an EDITING Phrase and the called procedure uses the APPLY statement, APPLY causes the appropriate action.

- If you are using APPLY in an EDITING Phrase and *expression* is a key that causes a GO action (GO key, RETURN key on the last field of a frame, or any key in a list used with the GO-ON statement), PROGRESS does not immediately exit the EDITING Phrase but instead processes all the remaining statements in the phrase. If RETRY, NEXT, UNDO RETRY, or UNDO NEXT is executed before the end of the phrase, PROGRESS ignores the GO and continues processing the Editing Phrase.

- PROGRESS ignores an APPLY statement outside an EDITING Phrase unless the APPLY statement is applying one of HELP, END-ERROR, ERROR, ENDKEY, BELL, or STOP.

- APPLY(-2) is the same as APPLY(ENDKEY)

SEE ALSO EDITING Phrase, GO-PENDING Function, KEYCODE Function, LASTKEY Function

ASC Function

Converts a character expression representing a single character into the corresponding ASCII integer value.

SYNTAX

```
ASC(expression )
```

expression

A constant, field name, variable name or any combination of these whose value is a single character that you want to convert to an ASCII value. If *expression* is a constant, you must enclose it in quotation marks (""). If the value of *expression* is other than a single character, ASC returns the value -1.

Uppercase and lowercase is significant for *expression*. For example, ASC("a") returns a different value than ASC("A").

EXAMPLE

r-asc.p

```
DEFINE VARIABLE ltrl AS INTEGER EXTENT 27.
DEFINE VARIABLE i AS INTEGER.
DEFINE VARIABLE j AS INTEGER.

FOR EACH customer:
  i = ASC(SUBSTRING(name,1,1)).
  IF i < ASC("A") or i > ASC("Z") THEN i = 27.
  ELSE i = i - ASC("A") + 1.
  ltrl[i] = ltrl[i] + 1.
END.
DO j = 1 TO 27 WITH NO-LABELS.
  IF j <= 26
  THEN DISPLAY CHR(ASC("A") + j - 1) @ ltr-name
    AS CHARACTER FORMAT "x(5)".
  ELSE DISPLAY "Other" @ ltr-name.
  DISPLAY ltrl[j].
END.
```

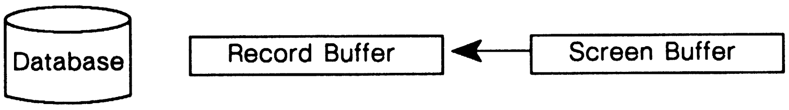
This procedure counts how many customer's names begin with each of the letters, capital A through capital Z. All other customers are counted separately. The ASC function is used to translate a letter into an integer which is then used as an array subscript for the counting.

SEE ALSO CHR Function, INTEGER Function

ASSIGN Statement

Moves data previously placed in a screen buffer, usually by a PROMPT-FOR statement, to the corresponding fields and variables.

DATA MOVEMENT



SYNTAX

$$\text{ASSIGN} \quad \left\{ \begin{array}{l} \textit{field} \\ \textit{field} \;=\; \textit{expression} \end{array} \right\} \ldots$$

ASSIGN *record* [EXCEPT *field* ...]

field
The name of the field to be set from its corresponding value in the screen buffer.

expression
A constant, field name, variable name, or any combination of these that results in a value you want to assign to the field you are naming. In this case, the value of the field is determined from the expression rather than from screen data.

record
The name of the record buffer, all of whose fields are to be set from the corresponding values in the screen buffer. Naming a record is a shorthand way of listing each field in that record individually.

To use ASSIGN with a record in a file defined for multiple databases, you may need to qualify the record's filename with the database name. See the description of the Record Phrase for more information.

EXCEPT *field*
All fields except those fields listed in the EXCEPT phrase are affected.

EXAMPLES

```
                                                          r-asgn.p
REPEAT:
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num NO-ERROR.
  IF NOT AVAILABLE customer
  THEN DO:
    CREATE customer.
    ASSIGN cust-num.
  END.
  UPDATE customer WITH 2 COLUMNS.
END.
```

This procedure prompts you for a customer number and retrieves the customer record if one exists, or creates a new one if it does not exist. If it creates a new record, the value for the cust-num field is ASSIGNed from the value you entered in response to the PROMPT-FOR statement.

```
                                                          r-asgn2.p
DEFINE VARIABLE neword LIKE order-line.order-num
                LABEL "New Order".
DEFINE VARIABLE newordli LIKE order-line.line-num
                LABEL "New Order Line".
REPEAT:
  PROMPT-FOR order-line.order-num line-num.
  FIND order-line USING order-line.order-num AND line-num.
  SET neword newordli.
  FIND order WHERE order.order-num = neword.
  ASSIGN order-line.order-num = neword
         order-line.line-num = newordli.

END.
```

This procedure changes the order number and line number of an order-line record. Essentially, it “copies” an order-line from one order to another. The new values are set into variables and the record is modified with a single ASSIGN statement containing two assignment phrases of the form *field = expression*. Thus, both fields are changed within a single statement. Because records are reindexed at the end of any statement that changes an index field value, and because order-num and line-num are used jointly in one index, this method avoids having the indexing done until both values are changed.

NOTES

- You usually use an ASSIGN statement after a PROMPT-FOR statement. PROMPT-FOR brings the value of a field or variable into a screen buffer. ASSIGN moves the value from the screen buffer into the field or variable.

- You will often use the PROMPT-FOR statement to get one or more index fields from the user, and the FIND statement to find a record matching those index values. If no record is found, you can then use the CREATE statement to create a new record, and the ASSIGN statement to assign to the new record the values the user supplied.

- You cannot use the SET statement in place of the PROMPT-FOR statement. The SET statement prompts the user for input and then assigns that input to the record in the record buffer. However, there may not be a record available for which SET can assign values.

- ASSIGN does not move data into a field or variable if there is no data in the corresponding screen field. There is data in a screen field if a DISPLAY of the field was done or if data was entered into the field. If you PROMPT-FOR a field or variable that has not been DISPLAYed in the frame and key in blanks, the field or variable is not changed because the screen field is considered changed only if the data differs from what was in the field.

- If a field or variable referenced in an ASSIGN statement is used in more than one frame, then the value in the frame most recently introduced in the procedure is used.

- If you type blanks into a field in which data has never been displayed, the ENTERED function returns FALSE and the SET or ASSIGN statement does not update the underlying field or variable. Also, if PROGRESS marks a field as entered, and the PROMPT-FOR statement prompts for the field again and you do not enter any data, PROGRESS no longer considers the field entered.

- If you use a single qualified identifier with the ASSIGN statement, the compiler first interprets the reference as *dbname.filename*. If the compiler cannot resolve the reference as *dbname.filename*, it tries to resolve it as *filename.fieldname*.

- You can use the Data Dictionary to let PROGRESS help you generate one type of ASSIGN statement: an assignment to copy some subset of a record's fields from one record to another. Enter the Data Dictionary. Press **U** (Utilities), **G** (Generate Include Files), and **A** (ASSIGN statement). Select a file with the arrow keys or by typing its first few letters, and then press RETURN. Specify a name for the include file; it is named *file-name*. i by default. For example, below is an include file that includes an ASSIGN statement for the fields in the file `order-line` from the PROGRESS `demo` database. Now you can use the PROGRESS editor, or any text editor, to delete one or more fields, to replace the arguments with *database.file* qualifiers, or to make any other changes.

```
                                                   order-ln.i
/* 01/01/90 COPY assignment */

ASSIGN
  {1}.Order-num = {2}.Order-num
  {1}.Line-num  = {2}.Line-num
  {1}.Item-num  = {2}.Item-num
  {1}.Price     = {2}.Price
  {1}.Qty       = {2}.Qty
  {1}.Qty-ship  = {2}.Qty-ship
  {1}.Disc      = {2}.Disc.
```

- Suppose you delete the `Order-num` line from the above include file. Now, the following line, inserted in any procedure, assigns the values of all fields but `Order-num` from an `order` file record in one database to the corresponding record fields in an `order` file in a second database.

```
{order-ln.i  "db2.order"  "db1.order"}
```

SEE ALSO = Assignment Statement, INPUT Function, PROMPT-FOR Statement, SET Statement

AVAILABLE Function

When you use the FIND statement or the FOR EACH statement to find a record, PROGRESS reads that record from the database into a record buffer. This record buffer has the same name as the file used by the FIND or FOR EACH statement unless you specify otherwise. The CREATE statement creates a new record in a record buffer. The AVAILABLE function returns a TRUE value if the record buffer you name contains a record and returns a FALSE value if the record buffer is empty.

SYNTAX

```
AVAILABLE record
```

record
The name of the record buffer you want to check.

To access a record in a file defined for multiple databases, you may need to qualify the record's filename with the database name. See the description of the Record Phrase for more information.

EXAMPLE

r-avail.p

```
REPEAT
  PROMPT-FOR item.item-num.
  FIND item USING item-num NO-ERROR.
➤ IF AVAILABLE item
  THEN DISPLAY idesc prod-line cost.
  ELSE MESSAGE "Not found".
END.
```

Here, the FIND statement with the NO-ERROR option bypasses the default error checking and the message you get when a record does not exist. Because the item-num is unique there is no need to use the AMBIGUOUS function to further pinpoint the cause of a record's not being AVAILABLE.

SEE ALSO AMBIGUOUS Function, FIND Statement, LOCKED Function, NEW Function, FOR Statement

BEGINS Function

Tests a character expression to see if that expression begins with a second character expression.

SYNTAX

```
expression1 BEGINS expression2
```

expression1

A constant, field name, variable name, or any combination of these whose character value you want to test to see if it begins with *expression2*.

expression2

A constant, field name, variable name, or any combination of these whose character value you want to compare to the beginning of *expression1*.

EXAMPLES

```
r-bgns.p

DEFINE VARIABLE cname LIKE customer.name LABEL "Name".

REPEAT:
  SET cname WITH SIDE-LABELS.
  FOR EACH customer WHERE name BEGINS cname:
    DISPLAY name address city st zip.
  END.
END.
```

In this example, the user supplies a customer name or the first characters of a customer name. The procedure finds customer records where the name field begins with the user's input. If the customer file is indexed on the name field, this procedure is very efficient and retrieves only the selected records.

The following procedure lists exactly the same customers but is much less efficient since it retrieves and examines all customer records, only displaying the ones with the appropriate names.

```
r-bgns2.p
```

```
DEFINE VARIABLE cname LIKE customer.name LABEL "Name".

REPEAT:
  SET cname WITH SIDE-LABELS. /* create MATCHES pattern */
  cname = cname + "*".
  FOR EACH customer WHERE name MATCHES cname:
    DISPLAY name address city st zip.
  END.
END.
```

NOTES

- BEGINS is especially useful in a WHERE phrase specifying which records should be retrieved in a FOR EACH block. In contrast to the MATCHES function, which requires that all records in the file be scanned, BEGINS uses an index wherever possible.

- Most character comparisons are case-insensitive in PROGRESS. By default, all characters are converted to upper case prior to comparisons. However, it is possible to define fields and variables as case sensitive (though it is not advised, unless strict ANSI SQL adherence is required). If either of the character expressions passed to BEGINS is a field or variable defined to be case-sensitive, the comparison is case-sensitive. In a case-sensitive comparison "SMITH" does not equal "Smith."

- Trailing blanks are considered in the BEGINS Function. For example,

```
"x"  BEGINS  "x      "
```

  is false. This is different than comparisons, where trailing blanks are ignored. For example, the following statement is true.

```
"x"  = "x      "
```

SEE ALSO MATCHES Function

BELL Statement

Causes the terminal to "beep."

SYNTAX

```
BELL
```

EXAMPLE

```
                                                          r-bell.p
    DEFINE VARIABLE outfile AS CHARACTER
           FORMAT "x(8)" LABEL "Output file name".
    getfile:
    DO ON ERROR UNDO, RETRY:
      SET outfile WITH SIDE-LABELS.
      IF SEARCH(outfile) = outfile
      THEN DO:
        MESSAGE "A file named" outfile already exists
                 in your directory".
        MESSAGE "Please use another name".
  ➤     BELL.
        UNDO getfile, RETRY getfile.
      END.
    END.
    OUTPUT TO VALUE(outfile).
    FOR EACH customer:
      DISPLAY name max-credit.
    END.
```

This procedure dynamically determines the output file to use for a report that lists all customer records. The SET statement gets the name of a file from the user. The SEARCH function returns an unqualified file name if that file already exists in your working directory. If the file exists in your working directory, messages are displayed, the work done in the DO block is undone, and the user can try another file name. (The procedure is only concerned with whether the file exists in your working directory. If SEARCH returns a directory other than your current working directory, you receive no messages and your work is not undone.) After you type a file name that does not already exist, the OUTPUT TO statement directs the output of the procedure to that file.

NOTES

- If the terminal is not the current output device, BELL has no effect.

BTOS Statement

Runs a program, BTOS command, BTOS submit file, or starts the BTOS executive to allow interactive processing of BTOS commands.

SYNTAX

```
                 [ btos-command                                                 ]
                 [ OS-APPEND file-expression-from    file-expression-to         ]
                 [ OS-COPY  file-expression-from     file-expression-to         ]
BTOS [SILENT]    [ OS-DELETE filename-expression...                             ]
                 [ OS-RENAME oldname-expression newname-expression              ]
                 [ OS-REQUEST Cd Erc nC nRq nRs C1..Cn Rq1..Rqn Rs1..Rsn         ]
                 [  [run-file  [command   [argument            ]      ]         ]
                 [  < run-file [          [VALUE(expression)   ] ...  ]         ]
                 [ SUBMIT submit-file-spec [parameter-list ...  ]               ]
```

[SILENT]
Turns off the pause between screens that occurs while running an operating system command. It also eliminates the pause that occurs before returning control to the calling PROGRESS procedure.

btos-command
Any BTOS command that is executed by a run file. When executed BTOS displays the command form and allows you fill in the form and press GO, the command is executed.

OS-APPEND
Appends a file, specified by *file-expression-from,* to the end of another file, specified by *file-expression-to.*

OS-COPY
Copies a file, specified by *file-expression-from,* to the end of another file, specified by *file-expression-to.*

OS-DELETE
Deletes the file specified the *filename-expression.* One of more files can be specified.

OS-RENAME

Renames the file specified by *oldname-expression,* to the name specified by *newname-expression.*

run-file

The name of the BTOS run file to be executed. The run file must start with a volume, "[", or directory, "<" specification.

command

The command name passed to the run file.

argument

One or more arguments you want to pass to the program being run by the BTOS statement. These arguments are expressions that PROGRESS converts to character values, if necessary.

VALUE*(expression)*

expression is a constant, field name, variable name, or any combination of these, whose value is an argument you want to pass to the program or submit file.

SUBMIT

Start the submit file specified by the *submit-file-spec* and passes the parameters specified by the *parameter-list.*

submit-file-spec

The name of the BTOS submit file to be run.

parameter-list

One or more parameters you want to pass to the submit file being run. These arguments are expressions that PROGRESS converts to character values, if necessary.

OS-REQUEST

Builds a request block using given variables, sends the request, and waits for a response.

NOTE: Requests provide low level system communications. Incorrect or improper use can cause unpredictable results, including system crashes. Refer to your CTOS Concepts manual for information on using requests.

Cd

BTOS/CTOS request code. This must be an integer constant less than 65535.

Erc

Name of a shared integer variable. The value of this variable is the returned error code. (Zero, if no error.)

nC

The number of integer constants that make up the control information part of the request block. Set *nC* to 3 if 6 bytes of control information is required.

nRq

The number of request pbcb pairs in the request block.

nRs

The number of response pbcb pairs in the request block.

C1..Cn

The integer constants or variable names that make up the control information. Variables used for control information must be shared integer variable. If *nC* is zero do not enter any control values.

Rq1..Rqn

The request values to send. Each value contains two parts, the data type and a shared variable name. Data types start with a percent sign (%), followed by a letter ("c", "i", "u", or "l"), and an optional subscript in brackets ([n]). Character variables must include a subscript n, which must be larger than the variables maximum length. Numeric subscripts indicate that the variable is an array. All elements of an integer array are concatenated together to create the request value. The shared integer arrays must have an extent value equal to the subscript value.

Unknown integer values sent as zeros. Unknown character values are sent as zero length strings.

Valid data types are:

%c[n]	character	n bytes
%i	signed integer	2 bytes
%u	unsigned integer	2 bytes
%l	signed long	4 bytes

Rs1..Rsn

The response values received. Each value contains two parts: the data type and a shared variable name. Response data types have the same form as request data types.

RETURNING TO PROGRESS

While running an interactive session (BTOS command with no options) you can return to PROGRESS by using the command "Exit Executive" for BTOS, or "Finish Executive" for CTOS.

The "PROGRESS Exit" command also returns you to PROGRESS, but first pauses and prompts you to "Press space bar to continue".

When ending a submit file use "PROGRESS Exit.", if you want to see the results of the last command run.

Use "Exit Executive" followed by "Finish Executive" (to run on both BTOS and CTOS), if you do not want to pause before returning to PROGRESS.

EXAMPLE

r-btos.p

```
IF OPSYS = "UNIX" then UNIX ls.
   ELSE IF OPSYS = "msdos" then DOS dir.
   ELSE IF OPSYS = "os2" then OS2 dir.
   ELSE IF OPSYS = "vms" then VMS directory.
➡  ELSE IF OPSYS = "btos" then  BTOS
     "[sys]<sys>files.run"
ELSE DISPLAY OPSYS  "is an unsupported operating system".
```

If the operating system you are using is UNIX, this procedure runs the UNIX ls command. If the operating system is VMS, then the procedure runs the VMS directory command. If you are using DOS, this procedure runs the DOS dir command. If you are using OS/2, this procedure runs the OS2 dir command. If you are using BTOS then the [sys] < sys > files.run files command is executed. Otherwise, a message is displayed stating the operating system is unsupported.

NOTES

- See the CTOS Statement for examples on how to use OS Request.
- Running the above command in PROGRESS automatically switches contexts, runs the operating system file specified, and then swaps contexts back, returning to the PROGRESS procedure making the call. The procedure will continue to process on the next line.
- If you use the BTOS statement in a procedure, the procedure will compile on, for example, a UNIX system, and the procedure will run as long as flow of control does not pass through the BTOS/CTOS statement while running on UNIX. You can use the OPSYS function to return the name of the operating system on which a procedure is being run. Using this function enables you to write applications that are fully transportable between any PROGRESS supported operating system even if they use the DOS, OS/2, UNIX, VMS and BTOS/CTOS statements.
- You cannot run executive intrinsic commands (commands contained in EXEC.RUN, which do not have actual run files) using the "BTOS command" statement. These commands include:

```
APPEND              PATH                RUN FILE
COPY                PLAYBACK            SCREEN SETUP
CREATE DIRECTORY    RECORD              SET PROTECTION
CREATE FILE         REMOVE DIRECTORY    STOP RECORD
DELETE              RENAME              TYPE
LIST                RUN                 VIDEO
LOGIN
```

- To use the *"btos-command"*, *"run-file"*, or "SUBMIT" options you must have the Context Manager installed.
- You can also access the interactive BTOS executive by selecting the option "e" from the main menu of PROGRESS Help.

SEE ALSO OPSYS Function, UNIX Statement, VMS Statement, DOS Statement, CTOS Statement, OS2 Statement.

CALL Statement

Transfers control to a dispatch routine (PRODSP) which then calls a C function. You write the C function using PROGRESS' Host Language Call Interface (HLC). PROGRESS HLC consists of a collection of C functions that perform the following:

- Obtain data from PROGRESS shared variables and buffers.
- Set data in PROGRESS shared variables and buffers.
- Control screen modes.
- Provide PROGRESS-like messages in the message area at the bottom of the screen.

Using HLC, you can extend PROGRESS with your own C functions.

SYNTAX

```
CALL routine-identifier [ argument ]...
```

routine-identifier
The name the PRODSP dispatch routine uses to identify the C function to call.

argument
One or more arguments that you want to pass to the C function.

SEE ALSO The *3GL Interface Guide*, chapters 2–7.

CAN-DO Function

Compares the current userid with a list of users that have permission to access a specified file. Returns a TRUE value if the userid matches an entry in the list. You generally use the CAN-DO function to do security checking.

SYNTAX

```
CAN-DO ( idlist [, string] )
```

idlist

An expression (a constant, field name, variable name, or any combination of these) whose value is a list of one or more userids. If the expression contains multiple userids, you must separate the userids with commas. Do not insert blanks between the userids.

Table 4 shows the values you can use in *idlist*. You can use any combination of values to define *idlist*, and you must separate the values with commas.

Table 4: Values To Use For idlists

Value	Meaning
*	All users are allowed access
user	This user has access
!*user*	This user does not have access
*string**	Users whose ids begin with " *string* " have access
!*string**	Users whose ids begin with "*string* " do not have access

string

A character expression whose value is a string. The *string is* checked against *idlist*. If you do not enter *string,* the compiler inserts the USERID function, which is evaluated each time you run the procedure. If you use the USERID function and have more than one database connected, be sure to include the database name. For example, `USERID ("dbname")`.

EXAMPLES

The first example is based on an activity permission file named permission. The permission file is not included in your demo database. However, the records in that file might look something like this:

Activity	Can-Run
custedit	manager,salesrep
ordedit	manager,salesrep
itemedit	manager,inventory.
reports	manager,inventory,salesrep

In the procedure r-cando.p, the FIND statement reads the record with the activity "custedit" in the permission file. (This assumes that the activity field is defined as a unique primary index.) The CAN-DO function compares the userid of the user running the procedure with the list of users in the can-run field of the custedit record. If the userid is either "manager" or "salesrep", the procedure continues executing. Otherwise, the procedure displays a message and control returns to the calling procedure.

r-cando.p

```
DO FOR permission:
  FIND permission "custedit".
  IF NOT CAN-DO(permission.can-run)
  THEN DO:
    MESSAGE "You are not authorized to run this procedure".
    RETURN.
  END.
END.
```

In the next example, the CAN-DO function compares *userid* (the userid for the current user) against the values in *idlist*. The values in *idlist* include "manager" and any userids beginning with "acctg" except "acctg8". If there is no match between the two values, the procedure displays a message and exits.

r-cando2.p

```
IF NOT CAN-DO("manager,!acctg8,acctg*")
THEN DO:
  MESSAGE "You are not authorized to run this procedure.".
  RETURN.
END.
```

NOTES

- If *idlist* contains contradictory values, the first occurrence of a value in the list applies. For example, CAN-DO("abc,!abc*","abc") is TRUE, since the userid abc appears before !abc in *idlist*.

- If *idlist* is exhausted without a match, CAN-DO returns a value of FALSE. Therefore, !abc restricts abc and everyone else (including the blank userid, ""). To restrict abc only and allow everyone else, use !abc, *.

- A *userid* comparison against *idlist* is not case sensitive.

- In addition to the examples shown above, you can use the CAN-DO function to compare a *userid* other than that of the current user against the list of values in *idlist*. For example, to assign a department *userid* to users "smith" and "jones" when they start PROGRESS, you can prompt these users for a department *userid* and password. You can then compare the supplied information against a table of identifiers.

  If the values supplied by the user match those in the identifier table, you can define a global shared variable to be used for the entire PROGRESS session. The value of this variable is the department *userid*. With the CAN-DO function you can then compare *userid* (the value of the global shared variable) against the list of values in *idlist*.

- If you know the name of the global shared variable, you can define another variable with the same name and call subroutines directly.

- Userids are established by the USERID and SETUSERID functions, or by the Userid (-U) option and Password (-P) option. The userid can be either an operating system userid (on UNIX, BTOS, and VMS) or a userid stored in PROGRESS' _User file (on UNIX, DOS, BTOS, VMS, and OS/2).

- You receive a compiler error if you omit *userid* and one of the following conditions exists:

  - There is no database connected.

  - More than one database is currently connected.

SEE ALSO SETUSERID Function, USERID Function reference pages; Chapter 11 of the *Programming Handbook*

CAN-FIND Function

Returns a TRUE value if a record is found which meets the specified FIND criteria. CAN-FIND does not make the record available to the procedure. You typically use the CAN-FIND function within a VALIDATE option in a data handling statement like UPDATE.

SYNTAX

```
CAN-FIND ( [ FIRST ] record [ constant ]
           [ LAST  ]

    [ WHERE expression                   ]
    [ USING field [ AND field] ...       ] ... )
    [ OF record                          ]
    [ USE-INDEX index                    ]
```

FIRST

Returns TRUE if CAN-FIND locates a record that meets the specified criteria.

LAST

Returns TRUE if CAN-FIND locates a record that meets the specified criteria.

record

The name of the record whose existence you are checking.

To use CAN-FIND to locate a record in a file defined for multiple databases, you may need to qualify the record's filename with the database name. See the description of the Record Phrase for more information.

WHERE *expression*

Qualifies the record for which CAN-FIND searches. The expression must return a TRUE or FALSE value.

USING *field* [AND *field*]

One or more names of indexed fields you want to use to search for a record. The field you name in this argument must have been previously entered into a screen field, usually with a PROMPT-FOR statement.

OF *record*

Qualifies the records to use by relating the record to a record in another file.

constant

The file you want to use has a primary index; the *constant* is the value of the last component field of that index for the record you want.

USE-INDEX *index*
Identifies the index you want CAN-FIND to use to find a record. If you do not use this argument, PROGRESS selects an index to use based on the criteria specified with the WHERE, USING, OF, or *constant* arguments.

EXAMPLE

```
r-canfnd.p

REPEAT:
  CREATE customer,
  UPDATE cust-num name sales-rep
➤     VALIDATE(CAN-FIND(salesrep WHERE
             salesrep.sales-rep = customer.sales-rep),
             "Invalid sales rep -- please re-enter").
END.
```

In this example, the UPDATE statement uses the VALIDATE option to make sure that the salesrep entered matches one of the salesreps in the database. The VALIDATE option uses the CAN-FIND function to find a record where the initials match.

NOTES

- If you name more than one field as part of the selection criteria, the fields must be jointly indexed. For example, the following CAN-FIND function will not work with the demo database:

  ```
  CAN-FIND(customer WHERE cust-num = x AND name = y)
  ```

  The cust-num and name fields, although both indexed on the customer file, are not part of the same index.

- When using CAN-FIND, you may not have *any* selection criteria that depends on a non-indexed field.

SEE ALSO FIND Statement

CAPS Function

Converts any lowercase letters in a character string expression to uppercase letters, and returns the resulting character string.

SYNTAX

```
CAPS(expression)
```

expression
A constant, field name, variable name, or any combination of these that results in a character string.

EXAMPLE

```
                                              r-caps.p
   REPEAT:
     PROMPT-FOR customer.cust-num.
     FIND customer USING cust-num.
     UPDATE name address city st.
➤    customer.st = CAPS(customer.st).
     DISPLAY customer.st.
   END.
```

Here, the CAPS function converts the characters in the state field to uppercase. (Another way to do this is to use a FORMAT of !(2) for st. The ! format symbol indicates that any letters entered into the corresponding input position should be converted to uppercase.)

SEE ALSO LC Function

CHOOSE Statement

After you display data, the CHOOSE statement moves a highlight bar among a series of choices and selects a choice when you press GO (F1), RETURN, or enter a unique combination of characters.

SYNTAX

```
         { ROW field                                    }
CHOOSE   {                                              }
         { FIELD  field  [ HELP char-constant ]  ...    }

                      [ AUTO-RETURN          ]
                      [ COLOR color-phrase   ]
                      [ GO-ON (key-label)... ]  ... [ frame-phrase ]
                      [ KEYS char-variable   ]
                      [ NO-ERROR             ]
                      [ PAUSE expression     ]
```

ROW *field*

Tells CHOOSE to move a highlight bar among iterations of a down frame. *field* is the name of the field on which you want to position the highlight bar. The ROW option is useful for browsing through a set of records, although *field* does not have to refer to database records.

If you use the ROW option with the CHOOSE statement, you will probably want to use the SCROLL statement also.

If you use ROW, you can add a COLOR statement to control the video display highlighting. See the SCROLL statement examples.

FIELD *field...*

Tells CHOOSE to move a highlight bar among a set of fields or set of array elements in a frame. *field...* is the file record or array variable through whose fields or elements you want to move the highlight bar. The FIELD option is useful for building menus. You can also supply help for *field.*

HELP *char-constant*

Lets you provide help text for each field, which PROGRESS displays when HELP is pressed. You can only provide help messages (automatically) when CHOOSE is used to move through the fields in a record or the elements in an array.

AUTO-RETURN

Tells PROGRESS to use the selection when you enter a unique string of characters. When you use AUTO-RETURN and the user enters a unique string of characters, PROGRESS sets the value of LASTKEY to KEYCODE ("return").

NO-ERROR

Overrides default error handling by the CHOOSE statement, and return control to the procedure. If you do not use the NO-ERROR option, the CHOOSE statement rings the bell when the user presses an invalid key.

If you use the NO-ERROR option and the user presses an invalid key, the CHOOSE statement ends. At this point, you usually want to use the LASTKEY function to test the value of the last key the user pressed and then take the appropriate action.

KEYS *char-variable*

If you want to highlight a particular choice when entering a CHOOSE statement, or if you want to know what keys the user pressed to make a selection, you can use the KEYS option. When you use the KEYS option, you must give the name of a character variable, *char-variable*. If *char-variable* is initialized to one of the choices before entering the CHOOSE statement, that choice is highlighted. As the user presses keys to move the highlight bar, PROGRESS saves those keystrokes in *char-variable*. You can test the value of *char-variable* after the CHOOSE statement returns control to the procedure.

GO-ON *(key-label) ...*

Names *key-labels* for keys that cause CHOOSE to return control to the procedure. If you do not use the GO-ON option, CHOOSE returns control to the procedure when the user presses GO (F1), RETURN, END-ERROR (F4), or types a unique substring when AUTO-RETURN is in effect.

COLOR *color-phrase*

Specifies a video attribute or color for the highlight bar. Here is the syntax of *color-phrase*:

```
{ NORMAL
  INPUT
  MESSAGES
  protermcap-attribute
  dos-hex-attribute
  [ BLINK- ] [BRIGHT-] [ fgnd-color ] [ /bgnd-color    ]
  [ BLINK- ] [RVV- ] [ UNDERLINE- ][BRIGHT- ][ fgnd-color  ]
  VALUE(expression)                                          }
```

For more information about *color-phrase*, see the Color Phrase reference page.

PAUSE *expression*

The *expression* specifies a time-out period in seconds. If the user does not make a keystroke for the specified number of seconds, the CHOOSE statement times out and

ATTR-SPACE

returns control to the procedure. The timeout period begins before the user's first keystroke and is reset after each keystroke. If CHOOSE times out, the value of LASTKEY is -1. Typically, the timeout period is large and is intended to trap inactivity.

frame-phrase

Specifies the overall layout and processing properties of a frame. Here is the syntax of *frame-phrase*:

```
       [ ACCUM                                                 ]
       [ ATTR-SPACE                                            ]
       [ CENTERED                                              ]
       [           { [ DISPLAY ] color-phrase  }               ]
       [ COLOR     { PROMPT color-phrase       } ...           ]
       [ COLUMN  expression                                    ]
       [ n COLUMNS                                             ]
       [ DOWN                                                  ]
       [ expression DOWN                                       ]
       [ FRAME frame                                           ]
       [ NO-ATTR-SPACE                                         ]
WITH   [ NO-BOX                                                ] ...
       [ NO-HIDE                                               ]
       [ NO-LABELS                                             ]
       [ NO-UNDERLINE                                          ]
       [ NO-VALIDATE                                           ]
       [ OVERLAY                                               ]
       [ PAGE-BOTTOM                                           ]
       [ PAGE-TOP                                              ]
       [ RETAIN n                                              ]
       [ ROW  expression                                       ]
       [ SCROLL  n                                             ]
       [ SIDE-LABELS                                           ]
       [ TITLE [ COLOR color-phrase ]  expression              ]
       [ TOP-ONLY                                              ]
       [ WIDTH  n                                              ]
```

For more information about *frame-phrase*, see the Frame-Phrase reference page.

If there is any chance that your procedure will run on a spacetaking terminal, you will want to use the ATTR-SPACE option for a frame you are using with the CHOOSE statement. Omitting this option will cause the highlight bar to be missing.

EXAMPLE

```
r-chsmnu.p

DEFINE VARIABLE menu AS CHARACTER EXTENT 4 FORMAT "x(7)"
       INITIAL [ "Browse", "Create", "Update", "Exit" ].
DEFINE VARIABLE proglist AS CHARACTER EXTENT 4
       INITIAL [ "brws.p", "cre.p", "upd.p", "exit.p"].
DEFINE VARIABLE i AS INTEGER.

FORM "Use the sample strip menu to select an action."
      WITH FRAME instruc CENTERED ROW 10.

REPEAT:
  VIEW FRAME instruc.
  DISPLAY menu WITH NO-LABELS ROW 21 NO-BOX ATTR-SPACE
      FRAME f-menu CENTERED.
  HIDE MESSAGE.
➤ CHOOSE FIELD menu GO-ON (F5) AUTO-RETURN
       WITH FRAME f-menu.
  IF SEARCH(proglist[FRAME-INDEX]) = ?
  THEN DO:
    MESSAGE "The program" proglist[FRAME-INDEX] "does not
exist.".
    MESSAGE "Please make another choice.".
  END.
  ELSE RUN VALUE(proglist[FRAME-INDEX]).
END.
```

This procedure displays a strip menu with four choices. The procedure defines two arrays; one holds the items for selection on the menu, the other holds the names of the programs associated with the menu selections. The CHOOSE statement allows the user to select an item from the strip menu. PROGRESS finds the number (within the array) associated with the item selected and the program associated with that number in the proglist array. PROGRESS runs the program if it exists, and displays a message and allows the user to select another item if the program does not exist. (In your own application, you will associate actions with items selected by the CHOOSE statement.)

The GO-ON option makes the GET (F5) key behave like the GO (F1) key. With the LASTKEY function, you could check for the F5 key and take another action relevant to your application.

NOTES

- The CHOOSE statement takes different actions depending on which key you press and whether you use the NO-ERROR option, as shown in Table 5.

Table 5: CHOOSE Statement Actions

KEY	NO-ERROR	ACTION
Valid cursor motion [1]	N/A	Clear saved keys, move highlight bar
Invalid cursor motion [2]	NO	Clear saved keys, ring bell.
Invalid cursor motion [3]	YES	Clear saved keys, return control to procedure
A non-unique string followed by an alphanumeric character that does not make a matchable string [4]	NO	Clear saved keys, try to match the last key entered. If no match is available, ring bell.
A non-unique string followed by an alphanumeric character that does not form a matchable string with the other characters	YES	Return control to procedure
An invalid string	NO	Ring bell
An invalid string	YES	Return control to the procedure and, if the KEYS option was used, save any printable keys
Other keys [5]	NO	Ring bell
Other keys [6]	YES	Return control to procedure

[1] Valid cursor motion keys are CURSOR UP, CURSOR DOWN, CURSOR RIGHT, CURSOR LEFT, SPACEBAR, TAB, and BACKTAB when used to move the cursor within a frame.

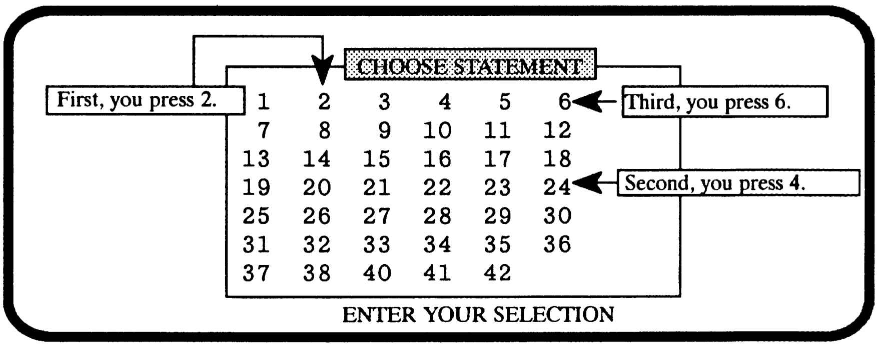
[2],[3] Invalid cursor motion keys are CURSOR UP, CURSOR DOWN, CURSOR RIGHT, and CURSOR LEFT, which cause the cursor to move outside the frame.

[4] The following example shows what the CHOOSE statement does when the user enters a non-unique string followed by a character that together with the rest of the string does not match anything:

r-chs1.p

```
DEFINE VARIABLE abc AS CHARACTER FORMAT "x(3)" EXTENT 42.
DEFINE VARIABLE i AS INTEGER.

DO i = 1 TO 42:
  abc[i] = STRING(i,">9").
END.

DISPLAY abc NO-LABELS WITH ATTR-SPACE CENTERED ROW 4
      TITLE " CHOOSE STATEMENT " FRAME f-choose WIDTH 36.
DISPLAY "Enter your selection " WITH CENTERED NO-BOX
      FRAME f-instruct.
PAUSE 1 BEFORE-HIDE NO-MESSAGE.

REPEAT:
  HIDE MESSAGE.
  CHOOSE FIELD abc AUTO-RETURN WITH FRAME f-choose.
  MESSAGE "You selected -> " FRAME-VALUE.
END.
```

When you run this procedure, your screen looks like this:



```
         CHOOSE STATEMENT
  1   2   3   4   5   6
  7   8   9  10  11  12
 13  14  15  16  17  18
 19  20  21  22  23  24
 25  26  27  28  29  30
 31  32  33  34  35  36
 37  38  40  41  42
      ENTER YOUR SELECTION
```

When you press 2, CHOOSE moves the highlight bar to 2. When you press 4, CHOOSE moves the bar to 24. When you press 6, CHOOSE looks for the string 246. Because it cannot find the string, it matches the last key pressed, 6, and places the highlight bar on 6.

[5], [6] Other keys are non-cursor-motion, non-alphanumeric keys (function keys, BACKSPACE) except for: HELP, STOP, RETURN, GO, END, ERROR, END-ERROR. Keys defined to do the actions of these keys still do so.

SEE ALSO Color Phrase, Frame Phrase, SCROLL Statement

CHR Function

Converts an ASCII integer value to its corresponding character value.

SYNTAX

```
CHR( expression)
```

expression

A constant, field name, variable name or any combination of these whose value is an integer that you want to convert to a character value.

If the value of *expression* is in the range of 1 to 255, CHR returns a single character. This character may not be printable or may not display on certain terminals. If the value of *expression* is outside that range, CHR returns a null string.

EXAMPLE

r-chr.p

```
DEFINE VARIABLE letter AS CHARACTER FORMAT "x(1)" EXTENT 26.
DEFINE VARIABLE i AS INTEGER.

DO i = 1 TO 26:
➤letter[i] = CHR((ASC("A") - 1) + i).
END.
DISPLAY SKIP(1) letter WITH 2 COLUMNS NO-LABELS
  TITLE "T H E  A L P H A B E T".
```

This procedure initializes the 26 elements of the letter array to the letters "A" through "Z".

SEE ALSO ASC Function, STRING Function

CLEAR Statement

Clears the data and colors (and side-labels for a down frame) displayed in a frame.

SYNTAX

```
CLEAR [ FRAME frame ] [ ALL ] [ NO-PAUSE ]
```

FRAME *frame*
: The name of the frame you want to clear. If you do not name a frame, CLEAR clears the default frame for the block containing the CLEAR statement.

ALL
: For a down frame (a frame used to display several occurrences of the fields in the frame), clears all occurrences and resets the current display position to the top of the frame.

NO-PAUSE
: Does not pause before clearing the frame. This is the default unless a DISPLAY has been done to the frame and you did not press a key after the display.

EXAMPLE

The r-clear.p procedure displays the PROGRESS data types and their corresponding default formats. The procedure asks if you want to enter values so you can see how PROGRESS formats those values. If you answer yes, PROGRESS clears the values currently displayed so you can enter your own values.

```
                                                      r-clear.p

DEFINE VARIABLE a AS CHARACTER INITIAL "xxxxxxxx".
DEFINE VARIABLE b AS DATE INITIAL TODAY.
DEFINE VARIABLE c AS DECIMAL INITIAL "-12,345.67".
DEFINE VARIABLE d AS INTEGER INITIAL "-1,234,567".
DEFINE VARIABLE e AS LOGICAL INITIAL yes.

DISPLAY "This illustrates the default formats for the"
        different data types" SKIP (2) WITH CENTERED
        ROW 4 NO-BOX FRAME head.
FORM "CHARACTER default format is ""x(8)""       " a SKIP
     "DATE default format is 99/99/99            " b SKIP
     "DECIMAL default format is ->>,>>9.99       " c SKIP
     "INTEGER default format is ->,>>>,>>9       " d SKIP
     "LOGICAL default format is yes/no           "
             e TO 55 SKIP
     WITH ROW 8 NO-BOX NO-LABELS CENTERED FRAME ex.
REPEAT:
  DISPLAY a b c d e WITH FRAME ex.
  MESSAGE "Do you want to put in some values?"
    UPDATE e.
  IF e THEN DO:
➤   CLEAR FRAME ex NO-PAUSE.
    SET a b c d WITH FRAME ex.
  END.
  ELSE LEAVE.
END.
```

NOTE

- A single (1-down) frame is automatically cleared whenever the block to which it is scoped iterates. A multi-frame (down frame) is automatically cleared whenever it is full and the block to which it is scoped iterates.

CLOSE Statement (SQL)

Closes an open cursor.

SYNTAX

```
CLOSE cursor-name
```

cursor–name
The name given to the cursor when defined in the DECLARE CURSOR statement.

EXAMPLE

```
ClOSE c01.
```

NOTES

- If you reopen the cursor, PROGRESS obtains the original retrieval set and repositions the cursor to the first row.
- The CLOSE statement can be used in both interactive SQL and embedded SQL.

Color Phrase

Specifies a video attribute or color.

SYNTAX

```
{ NORMAL                                                                  }
{ INPUT                                                                   }
{ MESSAGES                                                                }
{ protermcap-attribute                                                    }
{ dos-hex-attribute                                                       }
{ [ BLINK- ][BRIGHT-][fgnd-color ] [ / bgnd-color ]                       }
{ [ BLINK- ][RVV- ][ UNDERLINE- ][BRIGHT- ] [fgnd-color]                  }
{ VALUE(expression)                                                       }
```

For more information about protermcap see Chapter 2 of the *Programming Handbook*. For more information about the Video Options (-v) option (which uses DOS and OS/2 hex attributes), see Chapter 3 of the *System Administration II: General* guide.

INPUT, NORMAL, MESSAGES

The three standard colors PROGRESS uses for screen displays. NORMAL is used to display fields, INPUT is used to display input fields, and MESSAGES is used to display items in the message area.

If you are using DOS or OS/2, the default colors for NORMAL mode on a color monitor are a blue background and a white foreground; on a monochrome monitor the default colors are a standard background and foreground depending on the monitor. If you are using UNIX or VMS, NORMAL is whatever your terminal defines as normal display mode. If you are working on a BTOS/CTOS color monitor, the default colors for NORMAL mode are black background and a blue foreground.

On DOS and OS/2, the default colors for INPUT mode on a color monitor are a light gray background and a blue foreground; on a monochrome monitor the default is for input fields to be underlined. On UNIX and VMS, INPUT is dependent on the type of terminal and how INPUT is defined in the protermcap file, but is usually underlining. If you are working on a BTOS/CTOS, the default color for INPUT mode is yellow reverse video.

On DOS and OS/2, the default colors for MESSAGES on a color monitor are the same as for INPUT. On a monochrome monitor the default is reverse video. On UNIX or VMS, MESSAGES is dependent on the type of terminal and how MESSAGES is defined in the protermcap file, but is usually reverse video.If you are working on a BTOS/CTOS color monitor, the default colors for MESSAGES is red reverse video.

Under DOS and OS/2, you can change these defaults for an entire PROGRESS session by using the Video Options (-v) option when starting PROGRESS. Under UNIX, BTOS/CTOS, and VMS, the protermcap file supplied with PROGRESS supplies default attributes for NORMAL, INPUT, and MESSAGES for all defined terminals.

protermcap-attribute

You use the *protermcap-attribute* option only if you are using UNIX, BTOS/CTOS, or VMS. This is the name assigned to the attribute in the protermcap file (e.g. RED, BLINK, etc.). See Chapter 2 of the *Programming Handbook* for a description of the protermcap file.

dos-hex-attribute

A hex string whose value is 00 through FF.

[BLINK-] [BRIGHT-] [*fgnd-color*] [/*bgnd-color*]

Names specific colors you want to use for the screen foreground and background. You use this option only if you are using DOS or OS/2 and usually only if you are using a color monitor. Table 6 lists the colors you can use for *fgnd-color* and *bgnd-color*.

Table 6: DOS and OS/2 Colors

COLOR	ABBREVIATION
Black	Bla, Blk
Blue	Blu
Green	Gre, Grn
Cyan	C
Red	Red
Magenta	Ma
Brown	Bro, Brn
Gray	Gra, Gry
Dark-Gray	D-Gra
Light-Blue	Lt-Blu
Light-Green	Lt-Gre
Light-Cyan	Lt-C
Light-Red	Lt-Red
Light-Magenta	Lt-Ma
Light-Brown	Lt-Bro
Yellow	Y
White	W

If *fgnd-color* is omitted, then the foreground corresponding to NORMAL is used. If *bgnd-color* is omitted, then the background corresponding to NORMAL is used. If NORMAL, INPUT, or MESSAGES is specified for *fgnd-color* or *bgnd-color*, then the foreground or background color of the specified standard color is used.

[BLINK-] [RVV-] [UNDERLINE-] [BRIGHT-] [*fgnd-color*]

Names specific attributes you want to use for the screen display. You use this option only if you are using DOS or OS/2 and usually only if you are using a monochrome monitor. Normally you would never specify *fgnd-color.*

VALUE(*expression*)

An expression whose value results in one of the options in the COLOR phrase.

EXAMPLE

r-colphr.p

```
DEFINE VARIABLE hilite AS CHARACTER EXTENT 3.
DEFINE VARIABLE loop AS INTEGER.

hilite[1] = "NORMAL".
hilite[2] = "INPUT".    attribute to highlight
hilite[3] = "MESSAGES".

REPEAT WHILE loop <= 10:
  FORM bar AS CHARACTER WITH ROW(RANDOM(3,17))
       COLUMN(RANDOM(5,50)) NO-BOX NO-LABELS
       FRAME bursts.
➤ COLOR DISPLAY VALUE(hilite[RANDOM(1,3)]) bar
    WITH FRAME bursts.
  DISPLAY FILL("*",RANDOM(1,8)) @ bar WITH FRAME bursts.
  PAUSE 1 NO-MESSAGE.
  HIDE FRAME bursts NO-PAUSE.
  loop = loop + 1.
END.
```

This procedure displays, in 10 different occurrences, a random number of asterisks, in a random color, column, and row. The COLOR statement colors the display of the asterisks one of the three colors stored in the elements of the hilite array. The COLOR phrase in this example is VALUE(hilite[RANDOM(1,3)]. The result of this phrase is one of three colors in the hilite array. The DISPLAY statement uses the color determined in the COLOR statement to display a random number of asterisks.

NOTES

- The color-phrase entry is ignored for overlay frames on space-taking terminals.

SEE ALSO COLOR Statement

COLOR Statement

Indicates the video attribute or color to use for normal display, or to use when a field is ready for data entry.

SYNTAX

```
COLOR { [ DISPLAY ] color-phrase  } ... field ... [ frame-phrase ]
      {   PROMPT color-phrase     }
```

DISPLAY
Indicates that you want to use a specific color when a field is displayed.

PROMPT
Indicates that you want to use a specific color when a user is asked for input by an INSERT, PROMPT-FOR, SET, or UPDATE statement.

color-phrase
Specifies a video attribute or color. Here is the syntax of *color-phrase*:

```
{ NORMAL                                                          }
{ INPUT                                                           }
{ MESSAGES                                                        }
{ protermcap-attribute                                            }
{ dos-hex-attribute                                               }
{ [ BLINK- ][BRIGHT-][ fgnd-color ] [ / bgnd-color ]              }
{ [ BLINK- ][RVV- ][ UNDERLINE- ][BRIGHT- ][ fgnd-color ]         }
{ VALUE(expression)                                               }
```

For more information about *color-phrase*, see the COLOR-Phrase reference page. The color-phrase entry is ignored for overlay frames on space-taking terminals.

field
The name of the field or fields for which you want to override the default colors.

frame-phrase
Specifies the overall layout and processing properties of a frame.

Here is the syntax of *frame-phrase*:

```
       [ ACCUM                                                  ]
       [ ATTR-SPACE                                             ]
       [ CENTERED                                               ]
       [          { [ DISPLAY ] color-phrase       } ...        ]
       [ COLOR    { PROMPT color-phrase            }            ]
       [ COLUMN  expression                                     ]
       [ n COLUMNS                                              ]
       [ DOWN                                                   ]
       [ expression DOWN                                        ]
       [ FRAME frame                                            ]
       [ NO-ATTR-SPACE                                          ]
WITH   [ NO-BOX                                                 ] ...
       [ NO-HIDE                                                ]
       [ NO-LABELS                                              ]
       [ NO-UNDERLINE                                           ]
       [ NO-VALIDATE                                            ]
       [ OVERLAY                                                ]
       [ PAGE-BOTTOM                                            ]
       [ PAGE-TOP                                               ]
       [ RETAIN n                                               ]
       [ ROW  expression                                        ]
       [ SCROLL  n                                              ]
       [ SIDE-LABELS                                            ]
       [ TITLE [ COLOR  color-phrase     ] expression           ]
       [ TOP-ONLY                                               ]
       [ WIDTH  n                                               ]
```

For more information about *frame-phrase*, see the Frame Phrase reference page.

EXAMPLE

r-color.p

```
DEFINE VARIABLE hilite AS CHARACTER.
hilite = "messages". /* Use standard messages attribute
                        to highlight on-hand less than 50 */
FOR EACH item:
  DISPLAY item-num idesc on-hand WITH ATTR-SPACE.
  IF on-hand < 50 THEN COLOR DISPLAY
     VALUE(hilite) item-num on-hand.
END.
```

This procedure highlights the item number and on-hand fields for items whose on-hand value is less than 50. The variable hilite holds the video attribute (color) to do the highlighting. In this case, whatever attribute is used for the message area (such as reverse video, bright, or a color) is used.

NOTES

- When the output destination is not the terminal, PROGRESS disregards the COLOR statement.
- The COLOR statement does not automatically bring into view a frame on which a field's color attribute is changing.
- Use one of these statements to reset a field to the PROGRESS default colors: `COLOR DISPLAY NORMAL PROMPT INPUT` *field* or `COLOR DISPLAY NORMAL PROMPT INPUT` *field* `WITH FRAME` *frame*.
- If you run precompiled (.r) procedures on a spacetaking terminal, the frame field to which a color or other video attribute is applied must be specified explicitly, or by default, as ATTR-SPACE.
- If you write a procedure (for a nonspacetaking terminal) that uses color and you run it on a spacetaking terminal, PROGRESS does not display the colors. To display the colors, you must use the ATTR-SPACE option.
- Certain terminals, like the WYSE 75, are nonspacetaking for some attributes and spacetaking for others.
- Under UNIX, BTOS/CTOS, and VMS, if you specify a color or video attribute that is not defined for the terminal, normal display is used instead.

SEE ALSO COLOR Phrase, DISPLAY Statement, Frame Phrase

COMMIT OFF Statement (SQL)

Disables the SQL statement COMMIT WORK and enables the PROGRESS transaction management facilities.

SYNTAX

```
COMMIT OFF
```

EXAMPLE

```
COMMIT OFF.
```

NOTES

- Enter the command COMMIT OFF in the PROGRESS editor to disable the SQL statement COMMIT WORK and enable the PROGRESS transaction management facilities. **You cannot use this statement in a PROGRESS procedure.**

SEE ALSO COMMIT ON, COMMIT STATUS, COMMIT WORK, and ROLLBACK WORK statements.

COMMIT ON Statement (SQL)

Enables the SQL statement COMMIT WORK and disables the PROGRESS transaction management facilities.

SYNTAX

```
COMMIT ON
```

EXAMPLE

```
COMMIT ON.
```

NOTES

- Enter the command COMMIT ON in the PROGRESS editor to enable the SQL statement COMMIT WORK. **You cannot use the COMMIT WORK or COMMIT ON commands in a PROGRESS procedure.**

SEE ALSO COMMIT OFF, COMMIT ON, COMMIT STATUS, and ROLLBACK WORK statements.

COMMIT STATUS Statement (SQL)

Displays a message stating whether the SQL commit/rollback is enabled or disabled.

SYNTAX

```
COMMIT STATUS
```

EXAMPLE

```
COMMIT STATUS.
```

NOTES

- **You cannot use this statement in a PROGRESS procedure.**

SEE ALSO COMMIT OFF, COMMIT ON, COMMIT WORK, and ROLLBACK WORK statements.

COMMIT WORK Statement (SQL)

Commits all database changes affected by SQL data manipulation statements since the previous COMMIT WORK or ROLLBACK WORK statement or since the beginning of the session.

SYNTAX

```
COMMIT WORK
```

EXAMPLE

```
COMMIT WORK.
```

NOTES

- Because PROGRESS has its own transaction management facilities, the COMMIT WORK statement is disabled by default in interactive SQL. To enable it, enter the command COMMIT ON in the PROGRESS editor. **You cannot use the COMMIT WORK or COMMIT ON commands in a PROGRESS procedure.**

SEE ALSO COMMIT OFF, COMMIT ON, COMMIT STATUS, and ROLLBACK WORK statements.

COMPILE Statement

Compiles a procedure. After a procedure is compiled, the RUN statement does not recompile it, so the procedure runs quickly. A compilation can last for a session (a "session" compile) or can be saved permanently for use in later sessions (as an "object" or ".r" procedure).

SYNTAX

```
COMPILE { procedure              }  [ ATTR-SPACE    ]
        { VALUE ( expression )   }  [ NO-ATTR-SPACE ]

   [ XCODE expression                                                    ]
   [ SAVE [ INTO { directory | value(expression)} ]                      ]
   [                                                                     ] ...
   [ LISTING { listfile             } [ APPEND        ]                  ]
   [         { VALUE (expression )  } [ PAGE-SIZE n   ] ...              ]
   [                                  [ PAGE-WIDTHn   ]                  ]
   [ XREF filename [ APPEND ]                                            ]
```

procedure

The name of the procedure you want to compile. The procedure name (the last component of the full path name, if you are using the full path name) can be up to 12 characters long on a UNIX system; 8 characters long, plus an optional extension of up to three characters, on DOS and OS/2 systems; 39 characters long on VMS; up to 50 characters long under BTOS/CTOS. Under DOS, OS/2 and VMS, if you use the SAVE option then the procedure name must have an extension of .p, or no extension. Under UNIX, procedure names are case sensitive so you must enter the name exactly as it is stored.

VALUE (*expression*)

The value of a character expression which specifies the name of the procedure to be compiled or the name of the listing file.

ATTR-SPACE

ATTR-SPACE reserves spaces, in the frames used by a procedure, for field attributes like underlining and highlighting. See Chapter 7 of the *Programming Handbook* for more information about frames.

There are two ways a terminal can handle screen formatting. It can either:

- Reserve a character position on both sides of every field for special screen field attributes such as underlining or highlighting. These terminals are called "spacetaking" terminals.
- Not reserve a character position for special field attributes. These are called "nonspacetaking" terminals. All DOS and OS/2 terminals (monitors) are nonspacetaking.

NO-ATTR-SPACE

Does not reserve spaces, in frames used by a procedure, for field attributes such as underlining and highlighting. See Chapter 7 of the *Programming Handbook* for more information about frames.

XCODE *expression*

Decrypts the source code in *procedure*, and any encrypted include files, using the description key *expression*. Only use XCODE to decrypt files not encrypted with the default key. It is assumed the procedure and its include files were encrypted with the same key using the xcode utility supplied in the PROGRESS Developer's Toolkit. Include files that are not encrypted are included and compiled normally. The XCODE and LISTING options cannot be used together.

Decryption is incremental during compilation. Having the decryption key does not enable you to examine a decrypted version of the source code.

SAVE

Produces an object file which contains the object code for the procedure you are compiling. This object file is saved across PROGRESS sessions. If you do not use the SAVE phrase, the COMPILE statement produces object code for the source procedure, but the object code is not saved across PROGRESS sessions. This code is called a session compile version of the procedure.

If you use the SAVE option under DOS or OS/2, the procedure name must have an extension of .p or no extension.

The COMPILE SAVE statement produces an object file with the name*procedure-name*.`r`. If you supply a filename of `test`, COMPILE SAVE produces an object file with the name `test.r`. If you supply a filename of `test.p`, COMPILE SAVE produces an object file with the name `test.r`. Under UNIX and BTOS, the procedure `test.XXX` compiles into `test.XXX.r`. Under DOS, OS/2 and VMS, the file name `test.xxx.r` is not valid, so PROGRESS cannot save the compilation until you rename the file.

By default, the object file is stored in the same directory as the source procedure. If you use the SAVE INTO phrase, the .r file produced by a compilation can be saved in a different directory. See example and notes below for more information.

On UNIX, BTOS/CTOS,DOS, and OS/2, a newly created object file replaces any existing object files of the same name. On VMS, no previous versions of the object file are deleted and the new object file is given the next higher version number.

LISTING

Produces a listing of the compilation that includes the following:

- The name of the file containing the procedure you are compiling.
- The database name.
- The date and time at the start of the compilation.
- The number of each line in the procedure.
- The number of the block to which each statement belongs.
- The complete text of all include files and the name of any subprocedures.

listfile

The name of the file in which you want to store the compiler listing.

APPEND

Appends the current listing to the contents of the listing file. If you do not use the APPEND option, PROGRESS creates a new listing file, replacing any file of the same name.

PAGE-SIZE *n*

Identifies the number of lines to a page in the listing file. 55 is the default page size and *n* must be between 10 and 127.

PAGE-WIDTH *n*

Identifies the number of columns to a page in the listing file. 80 is the default page width and *n* must be between 80 and 255. It is a good idea to add at least 12 spaces to the page width in force when the procedure was typed; this allows for listing information that precedes each line of code, thereby insuring that the procedure appears in the listing output exactly as you typed it.

XREF *filename*

Writes cross-reference information between procedures and database objects to the file *filename*. XREF generates one unformatted, blank-separated line in *filename* for each object reference. Each line has the following format.

```
procedure-name file-name line-number reference-type object-identifier
```

The *procedure-name* is the name of the procedure you compile with the COMPILE XREF statement. The *file-name* is the name of the file whose code contains the *reference-type*.

The possible *reference-types* and *object-identifiers* are:

reference-type	*object-identifier*
RUN	*procedure-name | value(exp)*
INCLUDE	*include-file-name*
CREATE	*[database.]file [WORKFILE]*
DELETE	*[database.]file [WORKFILE]*
ACCESS	*{ [database.] file field [WORKFILE] } | { SHARED variable }*
REFERENCE	*{ [database.] file field [WORKFILE] } | { SHARED variable }*
UPDATE	*{ [database.]file field [WORKFILE] } | { SHARED variable }*
SEARCH	*[database.]file { index | RECID | WORKFILE }*
NEW-SHR-VARIABLE	*new-shared-variable*
GLOBAL-VARIABLE	*global-variable*
SHR-FRAME	*shared-frame*
NEW-SHR-FRAME	*new-shared-frame*
SHR-WORKFILE	*shared-workfile [LIKE [database.]file]*
NEW-SHR-WORKFILE	*new-shared-workfile [LIKE [database.]file]*

EXAMPLES

```
r-cmple.p

COMPILE ord-ent SAVE.
```

Here, PROGRESS creates an object version of the ord-ent procedure, naming it ord-ent.r. (The ord-ent procedure is not included in the sample procedures supplied with PROGRESS).

```
r-cmple2.p

COMPILE demo1 ATTR-SPACE SAVE.
```

In this example, PROGRESS compiles the demo1 procedure, reserving spaces in frame layouts for special field attributes and producing an object file, demo1.r, that can be used across PROGRESS sessions. The .r file is saved in the current directory. You can save the .r file in a different directory by using the SAVE INTO phrase. For example, on a UNIX system to save a .r file in /usr/sources, enter:

```
COMPILE demo1 ATTR-SPACE SAVE INTO /usr/sources.
```

The next example shows the effect of of include files on compilation listings.

r-incl.p

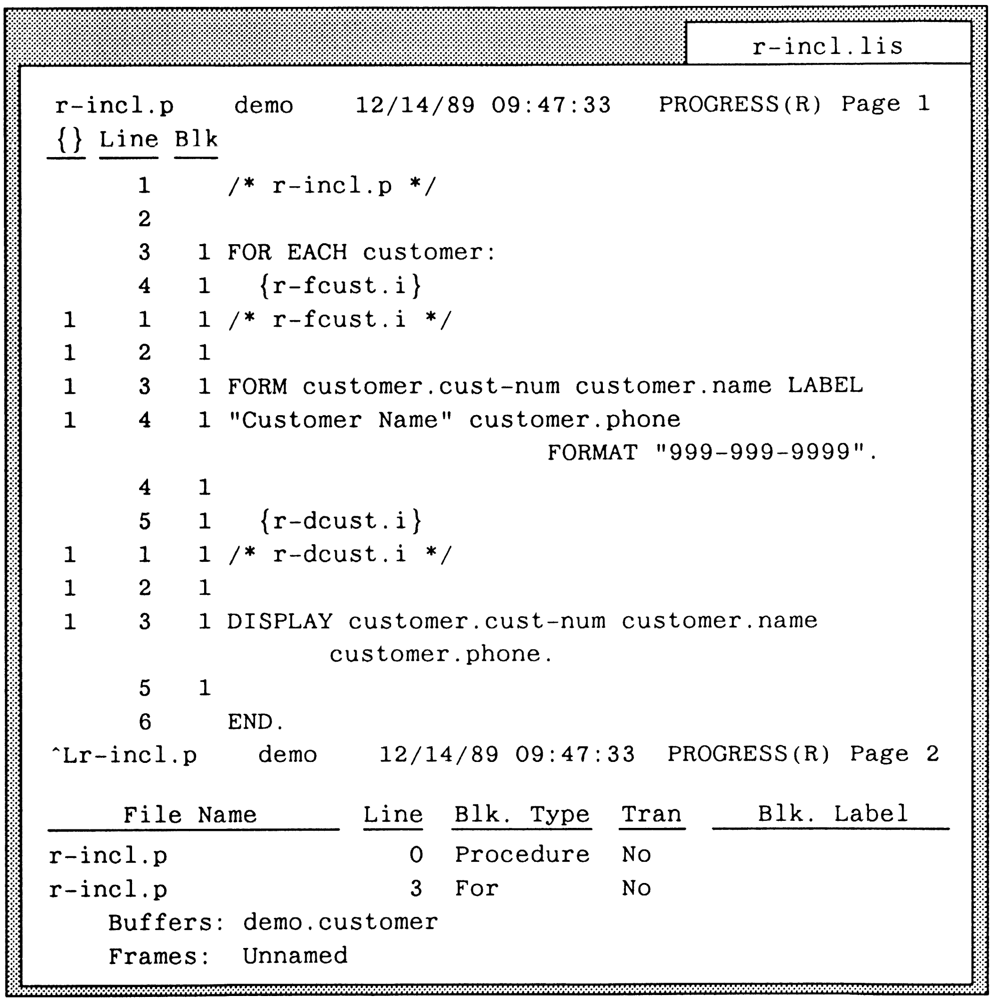
```
FOR EACH customer:
  {r-fcust.i}
  {r-dcust.i}
END.
```

Suppose you use the following COMPILE statement to compile the r-incl.p procedure:

r-comlis.p

```
COMPILE r-incl.p  SAVE  LISTING r-incl.lis  XREF r-incl.xrf.
```

This COMPILE statement produces three files: r-incl.r , r-incl.lis, and r-incl.xrf. Following are the contents of the r-incl.lis and r-incl.xrf files:



```
r-incl.p    demo    12/14/89 09:47:33   PROGRESS(R) Page 1
{} Line Blk
      1     /* r-incl.p */
      2
      3   1 FOR EACH customer:
      4   1   {r-fcust.i}
 1    1   1 /* r-fcust.i */
 1    2   1
 1    3   1 FORM customer.cust-num customer.name LABEL
 1    4   1 "Customer Name" customer.phone
                                 FORMAT "999-999-9999".
      4   1
      5   1   {r-dcust.i}
 1    1   1 /* r-dcust.i */
 1    2   1
 1    3   1 DISPLAY customer.cust-num customer.name
                   customer.phone.
      5   1
      6     END.
^Lr-incl.p    demo    12/14/89 09:47:33  PROGRESS(R) Page 2

     File Name        Line  Blk. Type  Tran    Blk. Label
r-incl.p                 0  Procedure  No
r-incl.p                 3  For        No
    Buffers: demo.customer
    Frames:  Unnamed
```

Due to the page formatting constraints in this manual, this sample output is not quite an exact copy of the `r-incl.lis` file.

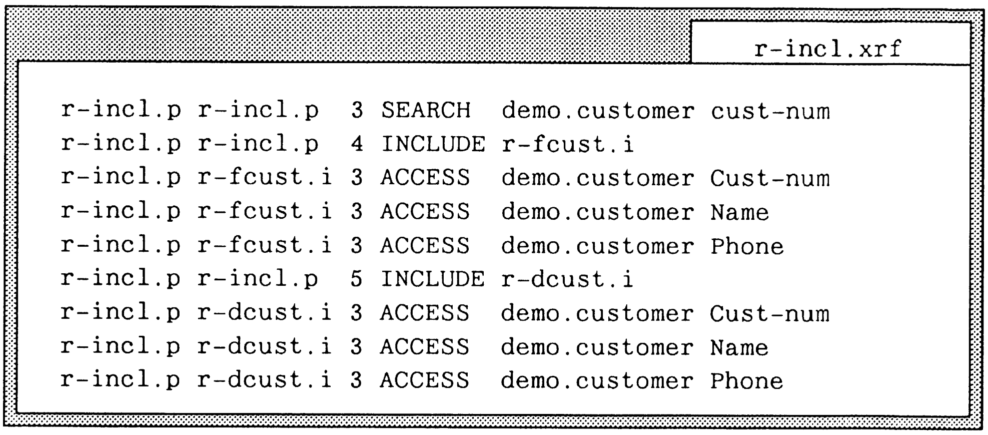
There are three columns next to the procedure in the listing file:

- {} - The level of the include file.
- Line - The line number in the procedure.
- Blk - The number of the block.

Following the procedure, there is information about each of the procedure blocks:

- Line - The line on which the block starts.
- Blk. Type - The type of block (Procedure, DO, FOR EACH, REPEAT).
- Blk. Label - The label of the block.
- Tran - Whether or not the block is a transaction block.
- Buf. Scope - The name of the record buffer scoped to the block.
- Frame Scope - The name of the frame scoped to the block.

The cross-reference file `r-incl.xrf`:



```
r-incl.p r-incl.p   3 SEARCH   demo.customer cust-num
r-incl.p r-incl.p   4 INCLUDE  r-fcust.i
r-incl.p r-fcust.i  3 ACCESS   demo.customer Cust-num
r-incl.p r-fcust.i  3 ACCESS   demo.customer Name
r-incl.p r-fcust.i  3 ACCESS   demo.customer Phone
r-incl.p r-incl.p   5 INCLUDE  r-dcust.i
r-incl.p r-dcust.i  3 ACCESS   demo.customer Cust-num
r-incl.p r-dcust.i  3 ACCESS   demo.customer Name
r-incl.p r-dcust.i  3 ACCESS   demo.customer Phone
```

Each line in the XREF file specifies the procedure, line number, access type, and access information. See the XREF option for a list of the values that follow a particular access type (*file* and *index* after SEARCH, for example). Also, see the sample procedure `xref.p` in the TOOLS subdirectory of the PROGRESS Procedure Library for an example of how to generate reports based on cross-reference information.

NOTES

- The value of the PROPATH environment variable defines the list of directories (path) to use when searching for the procedure file you name.

- If you are using UNIX, you typically define the PROPATH variable in a startup script or in your .profile file. If you are using DOS or OS/2, you may define the PROPATH variable in a .BAT file. If you are using BTOS/CTOS, you may define the PROPATH variable in a .ENV file. If you are using VMS, you may define the PROPATH variable in a .COM file, and you may use either a standard VMS directory specification or a UNIX directory specification. You can also define the PROPATH interactively at the operating system level.

  In addition to any directories you define for PROPATH, PROGRESS searches the directory containing the PROGRESS system software and the prodemo and proguide subdirectories. If you do not define a value for PROPATH, PROGRESS searches your working directory by default.

- To locate the source file that you name in the COMPILE SAVE statement, PROGRESS searches the first directory in PROPATH. If the source file is there, PROGRESS searches the same directory for an object file that was compiled against a different database than the one you are currently using. If such an object file exists, you receive an error message and the compilation stops. If no such object file exists, PROGRESS compiles the source file and creates an object file. On UNIX, BTOS, DOS, and OS/2, this new object file replaces any existing object file. On VMS, the new object file does not replace any existing files and it is given the next higher version number. If errors occur during compilation, no object file is produced and existing object files are left unchanged.

  If PROGRESS cannot find the source file, it searches for an object file. If a usable object file exists, you receive an error message and the compilation stops. If there is no object file, PROGRESS continues on to the next directory in PROPATH.

- Use the SAVE INTO phrase to store a compiled object file in a different directory than its corresponding source (`.p`) file. If you specify a relative pathname for the source file, that pathname is appended to the SAVE INTO path. For example (using UNIX pathnames):

```
PROPATH="/prol/source".
COMPILE  test/procl.p SAVE INTO /prol/obj.
```

  In the above example, the source file `/prol/source/test/procl.p` is saved as `/prol/obj/test/procl.r`

If the source file is a full pathname, *procedure-name*.r is stored in the SAVE INTO directory; its original directory path is dropped. For example:

```
COMPILE /prol/obj/test/procl.p SAVE INTO /usr/objects.
```

In this example, the source file is saved as /usr/objects/procl.r

- The ATTR-SPACE/NO-ATTR-SPACE designation in a Frame Phrase takes precedence over an ATTR-SPACE/NO-ATTR-SPACE designation in a Format Phrase. The ATTR-SPACE/NO-ATTR-SPACE designation in a Format Phrase takes precedence over an ATTR-SPACE/NO-ATTR-SPACE designation in a COMPILE statement.

- To locate the file you name with the COMPILE statement (without the SAVE phrase) PROGRESS searches the first directory in PROPATH for a usable object file. A usable object file must have:

  – The correct format: it must have been produced by the COMPILE SAVE statement.

  – Been produced by the current version of the PROGRESS compiler.

  – The same time stamp as any database files it references. When creating an object file, PROGRESS includes, as part of the object file, the time stamp of the most recent change to the database schema that affects **this** procedure (e.g., adding or deleting a field or index definition in a file the procedure references).

  – A length that is less than or equal to the length of the edit buffer as defined by the -e or /EDIT_BUFFER start-up option. This is the area in memory into which PROGRESS loads the object file before executing that file.

  – Read access to the object file, if you are using UNIX or VMS.

If there is a usable object file, there is no point in performing the compilation; you receive an error and the compilation stops. If PROGRESS does create a session compile version, the version is not used when you use the RUN statement. The RUN statement always uses an existing object file before using a session compile version of a procedure.

If there is no usable object file, PROGRESS searches the same directory in PROPATH for a source file. If the source file is there, PROGRESS compiles it into the session compile file. If it is not there, PROGRESS continues on to the next directory in PROPATH, searching for an object file first, then for a source file.

- In general, the PROGRESS object code produced on one brand of machine is not compatible with a different brand machine. However, code produced on a machine based on the Motorola 68000 CPU chip usually is compatible with other 68000-based machines.

- The size of the object code may vary, depending on the brand of machine on which it was compiled. If you have a very large procedure whose object code is at or very near to the limit of 63K, then that procedure may not compile on another machine. Differences between machines are due to differences in compiler and hardware alignment. For example, object code produced on an AT&T 3B series or Pyramid machine is usually 10 percent larger than that produced on 68000-based machines. The Edit buffer size (-e) option allows you to set the size of the edit buffer. For more information about this start up parameter and others, see Chapter 3 in the *System Administration II: General* guide.

- Modifications to existing field definitions do not affect database file time stamps. Therefore, updating a file's existing field definitions does not invalidate object versions of procedures that reference the file. However, adding or deleting files, fields, or indexes does affect database file time stamps and, therefore, invalidates object versions of procedures that reference the changed files.

- Both PROGRESS 4GL/RDBMS and PROGRESS Query/Run-Time include in the `prodemo` directory a procedure, `icompile.p`, which you can use to compile procedures. The `icompile.p` procedure builds and executes a COMPILE statement and can be used as an alternative to COMPILE, either interactively or in batch mode for compiling large numbers of procedures. See Chapter 13 in the *PROGRESS Language Tutorial* for a description of `icompile.p`.

SEE ALSO { } Include File, RUN Statement

CONNECT Statement

Allows access to one or more databases from within a PROGRESS procedure.

SYNTAX

$$\text{CONNECT}\left\{\begin{array}{l}[\textit{physical-name}]\\ [\textit{options}]\end{array}\right\}[\text{NO-ERROR}]$$

physical-name

The actual name of the database on disk. It can be a simple filename, or a fully qualified pathname, represented as an unquoted string, a quoted string, or a VALUE (*expression*).

options

One or more options, similar to those used to start PROGRESS. The allowed options (shown below) are actually a subset of PROGRESS startup options. See Chapter 3, of *System Administration II: General* for more information about PROGRESS startup options. Options are case-sensitive.

The following options are allowed in the CONNECT statement:

-1	-P	-U	-ct	-i
-B	-L	-pf	-a	-da
-F	-R	-c	-db	-n
-N	-S	-cl	-dt	-r
-H	-O	-cp	-g	-ld

NO-ERROR

Suppresses the error condition, but still displays the error message when an attempt to CONNECT to a database fails. You can use the CONNECTED function to determine whether the CONNECT succeeded.

EXAMPLE

```
connect.p

CONNECT mydb1 -1 -db mydb2 -1 NO-ERROR.
```

This procedure attempts to connect to databases mydb1 and mydb2 in single-user mode, with error suppression.

NOTES

- Each connected database is assigned a logical name for the current session, and is referred to by this logical name during the session. The `-ld` *logical name* option can be used to specify a logical name. If the logical name is not specified using the `-ld` option, then the physical database filename, minus the .db suffix, is the default logical name. For example, if the physical name is `/users/eastcoast/proapp/mydb.db`, then the default logical name is `mydb`. Logical names are **not** case-sensitive.

- You must connect to a given database before you run a procedure that references it. For example, assuming database `demo` has not been connected previously, `r-cnct1.p` below fails. At the start of execution `r-cnct1.p` checks whether `demo` is connected. If `demo` is not connected, a runtime error occurs. As is shown below, attempting to connect to `demo` within `r-cnct1.p` does not solve the problem.

```
/* r-cnct1.p */
/* NOTE: this code does NOT work */

CONNECT demo -1.
FOR EACH demo.customer:
  DISPLAY customer.
END.
```

Instead, split `r-cnct1.p` into two procedures, `r-cnct2.p` and `r-dispcu.p`:

r-dispcu.p

```
FOR EACH demo.customer:
  DISPLAY customer.
END.
```

r-cnct2.p

```
CONNECT demo -1.
RUN "r-dispcu.p".
```

This time, database `demo` is connected before `r-dispcu.p` is invoked, with the result that `r-dispcu.p` runs successfully.

- Databases can have aliases (see also ALIAS statement below). A database can have more than one alias, but each alias refers to only one database. The first database to be connected during a given session automatically receives the alias DICTDB. The first database to be connected that has an _MENU file automatically receives the alias FTDB. You can reassign the FTDB alias to any other FAST TRACK database.

- When you try to connect the same database twice using the same logical name, you get a warning which you can suppress by using NO-ERROR.

- When you try to connect different databases using the same logical name, you get an error message and an error condition. You can suppress the error condition by using NO-ERROR, and test using the CONNECTED function.

- When trying to connect to multiple databases and a connection fails, a run-time error occurs. Those databases that were successfully connected remain connected and program execution continues. Use the CONNECTED Function to find out which databases were successfully connected.

- If you run a procedure that requires a database and that database is not connected, PROGRESS searches for the database in the auto-connect lists in all connected databases. If PROGRESS finds the required database there, it automatically attempts to connect to the database using the parameters set for it in the auto-connect list. You can edit the auto-connect list using the database utilities in the PROGRESS data dictionary. If PROGRESS does not find it, then the connection attempt fails.

- Connection information contained in a PROGRESS auto-connect list for a given database is merged with connection information in a CONNECT statement that connects the database. That is, if you connect a database using a CONNECT statement, and that database has an entry in the PROGRESS auto-connect list of a previously connected database, the connection information in the auto-connect list entry and the connection information in the CONNECT statement will be merged. Connection information in the CONNECT statement takes precedence.

- You can connect to PROGRESS Version 5 databases as remote clients.

- On UNIX machines that do not support O_SYNC and SWRITE, permissions issues limit the use of the CONNECT statement for raw I/O connections to databases in single-user and multi-user direct access mode.

At startup, the `_progres` module has super-user privileges that allow it to open raw disk devices. Any databases specified on the startup command line can therefore be opened with raw I/O. After startup, the `_progres` module relinquishes the super-user privileges that allow it to open raw disk devices. As a result, you cannot use the CONNECT statement to establish a raw I/O connection to a database in either single-user or multi-user direct access mode.

If you attempt to use a CONNECT statement to open a raw I/O connection to a database in single-user mode, PROGRESS establishes a non-raw I/O connection to the database and displays a non-raw warning message.

When you attempt to use a CONNECT statement to open a raw I/O connection to a database in multi-user direct access mode, one of the following events occurs:

- If you started a server (PROSERVE) for the database with the non-raw I/O (-r) option, PROGRESS establishes a non-raw I/O connection to the database.
- If you started a server (PROSERVE) for the database with the raw I/O (-R) option, the CONNECT statement fails.

There are a number of ways to avoid these problems:

- Establish raw I/O database connections in the single-user and multi-user direct access modes at PROGRESS startup.
- If you must use the CONNECT statement to establish a raw I/O database connection, establish the connection with the multi-user client mode. Be sure to start the database server (PROSERVE) with the raw I/O (-R) startup option beforehand.
- If you must use the CONNECT statement to establish a raw I/O database connection in either single-user or multi-user direct access mode, do the following **CAREFULLY**:

  5. Change the permissions of the `_progres` module to `rwsrwsr-x` by typing **chmod 6775 _progres**.
  6. Change the group of the `_progres` module to match the group of the raw device (for example `/dev/rsd0d`) and block special device (for example, `/dev/sd0d`).
  7. Change the permissions of the raw and block special devices to `rw-rw----`.

A disadvantage of this procedure is that all files produced within PROGRESS have the same group as the disk devices.

– If you want to run a multi-user direct access session in non-raw mode, you must start the database server with the non-raw (-r) startup option.

If a database and accompanying before-image file have read-only permissions (`r--r--r--`) and you attempt to connect to that database in single-user mode using the CONNECT statement, the connection will fail with the error:

```
errno=13
```

This connection failure results from that fact that the `_progres` module relinquishes super-user privileges after startup and no longer possesses the privileges required to connect to the database using the CONNECT statement.

SEE ALSO DISCONNECT, CREATE ALIAS and DELETE ALIAS statements; CONNECTED, LDBNAME, PDBNAME, DBTYPE, DBRESTRICTIONS, , DBVERSION, GATEWAYS, FRAME-DB, NUM-DBS, and SDBNAME functions, and Chapter 13 of the *Programming Handbook*

CONNECTED Function

Lets you determine whether a database is connected. If *logical name* is the logical name or *alias* is the alias of a connected database, then the CONNECTED function returns TRUE; otherwise, it returns FALSE.

SYNTAX

$$\text{CONNECTED(}\left\{\begin{array}{l}\textit{logical-name}\\ \textit{alias}\end{array}\right\}\text{)}$$

logical-name
: Refers to a logical name. It can be a quoted string, or a character expression. An unquoted character string is not allowed here.

alias
: Refers to an alias. It can be a quoted string, or a character expression. An unquoted character string is not allowed.

EXAMPLE

```
r-cnctd.p

IF CONNECTED("demo") THEN RUN start.p.
```

This procedure runs `start.p`, if a database with the logical name demo is connected.

SEE ALSO CONNECT, DISCONNECT, CREATE ALIAS, and DELETE ALIAS statements; LDBNAME, PDBNAME, DBTYPE, DBRESTRICTIONS, GATEWAYS, FRAME-DB, and NUM-DBS functions.

COUNT-OF Function

Returns an integer that is the total number of selected records in the file or files you are using, across break groups.

SYNTAX

```
COUNT-OF ( break-group )
```

break-group

The name of a field or expression you named in the block header with the BREAK BY option.

EXAMPLE

r-cntof.p

```
FOR EACH customer BREAK BY st:
  DISPLAY cust-num name sales-rep st.
  ACCUMULATE st (SUB-COUNT BY st).
  IF LAST-OF(st)
➤ THEN DISPLAY 100 *  (ACCUM SUB-COUNT BY st st) /
        COUNT-OF(st)
               FORMAT "99.9999%"
               COLUMN-LABEL "% of Total!Customers".
END.
```

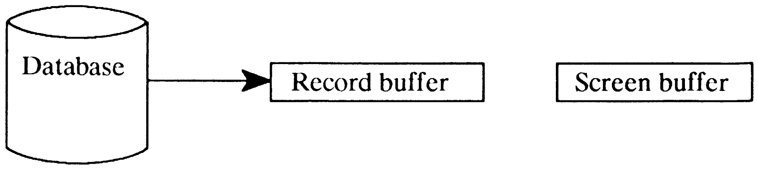
This procedure sorts all customers by state and then calculates the percentage of the total number of customers that are in each state. The COUNT-OF function provides the calculation with the number of customer records in the database.

SEE ALSO Aggregate Phrase (COUNT)

CREATE Statement

Creates a record in a file, sets all the fields in the record to their default initial values, and moves a copy of the record to the record buffer.

DATA MOVEMENT



SYNTAX

```
CREATE record [USING RECID ( n )]
```

record
The name of the record or record buffer you are creating.

To create a record in a file defined for multiple databases, you may need to qualify the record's filename with the database name. See the description of the Record Phrase for more information.

USING RECID (*n*)
Enables you to create a record for an RMS relative file using a specific record number, where *n* is the record-id of the record you want to insert.

EXAMPLE

r-create.p

```
  REPEAT:
➤   CREATE order.
    UPDATE order.order-num order.cust-num
      VALIDATE(CAN-FIND(customer OF order),
              "Customer does not exist") name odate.
    REPEAT:
➤     CREATE order-line.
      order-line.order-num = order.order-num.
      UPDATE line-num order-line.item-num
        VALIDATE(CAN-FIND(item OF order-line),
                "Item does not exist") qty price.
   END.
  END.
```

This procedure adds orders and order-lines to the database. Because the user supplies an order number when updating the order record, that order number is assigned (=) to the order-num field of the order-line record when the order-line record is created.

NOTES

- When you run procedures that create large numbers of records (e.g. during initial data loading), the process can run much faster if you use the -i or /INTEGRITY start-up option. See Chapter 3 of the *System Administration II: General* guide for more information on start-up options. (Back up your database before you use this option.)
- After you create a new record with CREATE, PROGRESS waits to write the record to the database until after the next statement that generates an index entry for the record.

SEE ALSO INSERT Statement, NEW Function

CREATE ALIAS Statement

Creates an alias for a database. The alias is added to a table of existing aliases.

SYNTAX

> CREATE ALIAS *alias* FOR DATABASE *logical name* [NO-ERROR]

alias

A synonym for *logical name*. It may be an unquoted string, a quoted string, or VALUE (*expression*). A database can have more than one alias, but each alias refers to only one database. Aliases can be used in place of the logical names to which they refer.

logical name

Refers to a previously set logical name. It can be an unquoted string, a quoted string, or a character expression. The logical name database must be connected unless NO-ERROR is used.

EXAMPLE

r-cralas.p

```
CREATE ALIAS myalias FOR DATABASE mydb NO-ERROR.
```

This procedure creates the alias `myalias` for database `mydb`.

NOTES

- The first PROGRESS database connected during a given session receives the DICTDB alias.
- The first database connected that has an _MENU file automatically receives the alias FTDB. You can reassign the FTDB alias to any other FAST TRACK database.
- If there is already a database connected with logical name equal to *alias*, CREATE ALIAS fails.
- If there is an existing alias equal to *alias*, the existing alias is replaced by the new alias.

- If you want to use an expression for an alias name or logical name, you must use CREATE ALIAS VALUE(*expression*) FOR DATABASE VALUE(*expression*).

- When a given database is disconnected, the existing aliases that refer to it are **not** erased, but remain in the session alias table. Later in the same session, if you connect to a database with the same logical name, the same alias is used again.

- The main purpose for aliases is to allow a general purpose application (such as the PROGRESS Database Dictionary) to expect a specific database name (the Dictionary only works on databases with logical name or alias "DICTDB"). The end-user or the application can use CREATE ALIAS to provide the correct alias, in case it is inconvenient to connect the database using the correct logical name. Also, if there are several connected databases, the application can ask the user which one to select, then set the alias accordingly. This is what the Dictionary does when you "Select Working Database".

- Suppose you connect to a database with logical name MYNAME and compile a procedure that accesses that database. Normally, the saved .r file contains references to MYNAME.

  In a later session, when you want to use the precompiled program, you may connect to your database with the same logical name (MYNAME), or you may connect with a different logical name and set up an alias with the statement CREATE ALIAS "MYNAME" FOR DATABASE *logical name*.

- Normally, any alias that exists during the session when you compile a procedure has no effect on the resulting .r file. When a procedure gets compiled, it is the logical name of the database that is accessed within the procedure that is put into the .r file, not any existing alias. If a procedure accesses more than one database, all of the logical names of accessed databases are placed into the .r file.

  However, there is an exception to the rule that only logical names are included in .r files. Any file reference that is qualified with an alias (as opposed to a logical name) generates a new instance of the file for the compilation. This new instance causes the r-code to have the alias reference and not the logical database name reference. Subsequent unqualified references to that same file within the same block, or nested blocks, will resolve to the new "alias instance" following the usual rules for qualifying. Unqualified references to different files in the same database do not get the alias name, but get the logical name. Anonymous references to a file, previously referenced using the alias qualifier, in a different, non-nested block do not get the alias name, but do get the logical name.

Naturally, it simpler to just connect to a database with the desired logical name, leave all references unqualified, not create an alias, and then compile the application. However, sometimes you cannot precompile. In those cases, if you wish to compile a procedure so that only the alias gets into the .r file, then explicitly qualify all file references using the alias. The reason for wanting only the alias to get into the .r file, is so that you can compile and distribute procedures that will run against any database whose logical name has been assigned the alias contained in the .r file.

- Changes made to an alias do not take effect within the current procedure. For example, `aliasl.p` below fails to compile when it reaches the FOR EACH statement, because alias `myalias` has not been created during the compilation.

```
/* aliasl.p */
/* NOTE: this code does NOT work */
   CREATE ALIAS myalias FOR DATABASE demo.
   FOR EACH myalias.customer:
     DISPLAY name.
   END.
```

To solve this problem, split `r-aliasl.p` into two procedures:

r-dispnm.p

```
/* r-dispnm.p */
   FOR EACH myalias.customer:
     DISPLAY name.
   END.
```

r-alias2.p

```
/* r-alias2.p */
  CREATE ALIAS myalias FOR DATABASE demo.
  RUN r-dispnm.p.
```

Another consequence of the fact that CREATE ALIAS effects only subsequent compilations is that currently executing procedures are not affected.

- Be careful when using shared buffers with aliases. If you reference a shared buffer after changing the alias that initially was used in defining it, you will get a runtime error. The following example illustrates:

r-main.p

```
/* r-main.p */
  CREATE ALIAS myalias FOR DATABASE demo1.
  RUN r-makebf.p.
```

r-makebf.p

```
/* r-makebf.p */
 DEFINE NEW SHARED BUFFER mybuf FOR myalias.customer.
 CREATE ALIAS myalias FOR DATABASE demo2.
 RUN r-disp6.p
```

r-disp6.p

```
/* r-disp6.p */
 DEFINE SHARED BUFFER mybuf FOR myalias.customer.
 FOR EACH mybuf:
   DISPLAY mybuf.
 END.
```

In this example, procedure `r-main.p` is run. It calls `r-makebf.p`, which in turn calls `r-disp6.p`. The alias `myalias` is created in `r-main.p`, with reference to database `demo1`. In `r-makebf.p`, the shared buffer `buf1` is defined for `myalias.friend`. Then, in the next line, `myalias` is changed, so that it now refers to database demo2. When an attempt is made to reference shared buffer `buf1` in procedure `r-disp6.p`, a run-time error occurs, with the message: " `r-disp6.p Unable to find shared buffer for mybuf` ".

SEE ALSO CONNECT, DISCONNECT, and DELETE ALIAS statements; CONNECTED, LDBNAME, PDBNAME, DBTYPE, DBRESTRICTIONS, GATEWAYS, FRAME-DB, and NUM-DBS functions.

CREATE INDEX Statement (SQL)

Creates an index.

SYNTAX

```
CREATE [UNIQUE] INDEX index-name ON table-name (column-list)
```

[UNIQUE]
: Indicates that all values in the specified column or combination of columns must be unique.

index-name
: The name you want to assign to the index. The index name must be unique within the database, not just within the table.

ON *table-name*
: The name of the table that contains the column or columns you want to index.

column-list
: The names of one or more columns to be indexed.

EXAMPLE

```
CREATE UNIQUE INDEX num
     ON cust_table (cust_num).
```

NOTES

- The index is built immediately for all existing rows in the table.
- If you specify UNIQUE and all values in the existing rows are not unique, PROGRESS/SQL returns an error message and the index is not created.

CREATE TABLE Statement (SQL)

Creates a new base table containing the columns you specify.

SYNTAX

```
CREATE TABLE table-name  ( { { column-name datatype
                              [ NOT NULL [ UNIQUE ]]
                              [ FORMAT string ] [ LABEL string ]
                              [ COLUMN-LABEL string [! string] ...]
                              [ [ NOT] CASE-SENSITIVE ]
                              [ DEFAULT initial value ] }
                            | { UNIQUE ( column-name [,...] ) }} [,...] )
```

table-name
: The name of the base table you want to create.

column-name
: The name of a column for the table.

datatype
: The data type of the column you are defining. The data types are: CHARACTER, INTEGER, SMALLINT, DECIMAL, FLOAT, DATE, and LOGICAL. See Chapter 4 for information about data types.

[NOT NULL]
: Indicates that the column must have a value.

[UNIQUE]
: Indicates that all values in the column must be unique. This keyword is an alternative to the UNIQUE clause, described below.

[FORMAT *string*]
: Specifies a new display format for the column. The *string* must be enclosed in either single or double quotation marks. See Chapter 4 for information about the various types of display formats.

[LABEL *label*]
: Specifies a new label for the column. The *label* must be enclosed in either single or double quotation marks. See Chapter 7 for information about labels.

[COLUMN-LABEL *label*]
Specifies a new label for the column when its values are displayed vertically (in columns) on the screen or in a printed report. The *label* must be enclosed in either single or double quotation marks. See Chapter 7 for information about column labels.

[NOT] CASE-SENSITIVE
Indicates whether the values in a character column and comparisons made to it are to be case-sensitive. The default is case-sensitive if you used the ANSI SQL (-Q) startup option. Otherwise, the default is not case-sensitive. (See the notes below that pertain to the -Q startup option).

[DEFAULT *initial-value*]
Assigns a default value for the column. This is the same as setting the initial value for a field in PROGRESS Data Dictionary.

UNIQUE (*column-name* [,...])
Indicates that all values in the column or combination of columns must be unique. If you use the UNIQUE option, PROGRESS/SQL automatically creates a unique index. The index is named `sql-uniq`*n*, where *n* is a number that makes the index name unique within the table. If you specify UNIQUE, you must also specify NOT NULL.

EXAMPLE

```
CREATE TABLE employee ( emp_num INTEGER NOT NULL UNIQUE,
                        name CHARACTER (30) NOT NULL,
                        job CHARACTER (15),
                        dept CHARACTER (15),
                        startdate DATE,
                        salary DECIMAL (8,2) ).
```

NOTES

- The user who creates a table can grant privileges on that table to other users. See the GRANT statement.
- You cannot use DROP INDEX to remove an index created with the UNIQUE option in a CREATE TABLE statement because the index is part of the base table definition. To create an index that you can later remove, use CREATE INDEX.
- You cannot use CREATE TABLE to create a table in an ORACLE database, you can, however, use it to create a table in a schema holder database.

- You should use the CASE-SENSITIVE option only when it is important to distinguish between uppercase and lowercase values entered for a character column. For example, you would need to use CASE-SENSITIVE to define a column for a part number that contains mixed uppercase and lowercase characters.

- If you use the ANSI SQL (-Q) option to enforce strict ANSI SQL conformance, all columns created by the CREATE TABLE statement are case-sensitive by default. Use the NOT CASE-SENSITIVE option for each column for which you want to override the default. See Chapter 3 of *System Administration II: General* for more information about the -Q startup option.

- When you use the -Q option to start a PROGRESS session, case-sensitivity is not enforced for columns that are already defined as being case-*in*sensitive.

SEE ALSO *System Administration II: General*

CREATE VIEW Statement (SQL)

Creates a viewed table (view) from one or more base tables and/or other views.

SYNTAX

> CREATE VIEW *view-name* [(*column-list*)] AS *SELECT-statement*
> [WITH CHECK OPTION]

view-name
The name of the view you want to create.

[*column-list*]
The names for the columns in the view. If you do not specify column names, PROGRESS/SQL uses the column names in the SELECT statement. If the select list includes expressions or duplicate column names, you must specify column names for the view.

AS *SELECT-statement*
Defines how to derive the data for the view.

[WITH CHECK OPTION]
Ensures that all updates to the view satisfy the view-defining condition in the WHERE clause of the SELECT statement.

EXAMPLE

```
CREATE VIEW doc
    AS SELECT emp_num, name, job
        FROM employee
        WHERE dept = 'Documentation'.
```

NOTES

- The CREATE VIEW statement can be used only in interactive SQL.
- You must have the SELECT privilege on all tables and views referenced by the view you are creating.
- You cannot use CREATE VIEW to create a view in an ORACLE database, you can, however, create a view in the schema holder that refers to ORACLE objects.
- The following restrictions apply to updating tables through views:
  - The view definition must not include the keyword DISTINCT, a GROUP BY clause, a HAVING clause, an expression, or an aggregate function.
  - The FROM clause of the SELECT statement must specify only one table.
  - The WHERE clause must not include a subquery.
  - If the FROM clause references a view, the view must be updatable.
  - Only an updatable view can include the keywords WITH CHECK OPTION in its definition.

CTOS Statement

Runs a program, CTOS command, CTOS submit file, or starts the CTOS executive to allow interactive processing of CTOS commands.

SYNTAX

```
               [ ctos-command                                                  ]
               [ OS-APPEND file-expression-from    file-expression-to          ]
               [ OS-COPY  file-expression-from     file-expression-to          ]
CTOS [SILENT]  [ OS-DELETE filename-expression...                              ]
               [ OS-RENAME oldname-expression newname-expression               ]
               [ OS-REQUEST Cd Erc nC nRq nRs C1..Cn Rq1..Rqn Rs1..Rsn         ]
               [  [run-file  [command     [argument            ]     ]         ]
               [  <run-file  [            [VALUE(expression)   ] ... ]         ]
               [ SUBMIT submit-file-spec  [parameter-list ...  ]               ]
```

[SILENT]
: Turns off the pause between screens that occurs while running an operating system command. It also eliminates the pause that occurs before returning control to the calling PROGRESS procedure.

ctos-command
: Any CTOS command that is executed by a run file. When executed CTOS displays the command form and allows you fill in the form and press GO, the command is executed.

OS-APPEND
: Appends a file, specified by *file-expression-from,* to the end of another file, specified by *file-expression-to.*

OS-COPY
: Copies a file, specified by *file-expression-from,* to the end of another file, specified by *file-expression-to.*

OS-DELETE
: Deletes the file specified the *filename-expression.* One of more files can be specified.

OS-RENAME

Renames the file specified by *oldname-expression,* to the name specified by *newname-expression.*

VALUE(*expression*)

expression is a constant, field name, variable name, or any combination of these, whose value is an argument you want to pass to the program or submit file.

SUBMIT

Start the submit file specified by the *submit-file-spec* and passes the parameters specified by the *parameter-list.*

submit-file-spec

The name of the BTOS submit file to be run.

run-file

The name of the CTOS run file to be executed. The run file must start with a volume, "[", or directory, "< " specification.

command

The command name passed to the run file.

argument

One or more arguments you want to pass to the program being run by the CTOS statement. These arguments are expressions that PROGRESS converts to character values, if necessary.

parameter-list

One or more parameters you want to pass to the submit file being run. These arguments are expressions that PROGRESS converts to character values, if necessary.

OS-REQUEST

Builds a request block using given variables, sends the request, and waits for a response.

NOTE: Requests provide low level system communications. Incorrect or improper use can cause unpredictable results, including system crashes. Refer to your CTOS Concepts manual for information on using requests.

Cd

BTOS/CTOS request code. This must be an integer constant less than 65535.

Erc

Name of a shared integer variable. The value of this variable is the returned error code. (Zero, if no error.)

nC

The number of integer constants that make up the control information part of the request block. Set *nC* to 3 if 6 bytes of control information is required.

nRq

The number of request pbcb pairs in the request block.

nRs

The number of response pbcb pairs in the request block.

C1..Cn

The integer constants or variable names that make up the control information. Variables used for control information must be shared integer variable. If *nC* is zero do not enter any control values.

Rq1..Rqn

The request values to send. Each value contains two parts, the data type and a shared variable name. Data types start with a percent sign (%), followed by a letter ("c", "i", "u", or "l"), and an optional subscript in brackets ([n]). Character variables must include a subscript n, which must be larger than the variables maximum length. Numeric subscripts indicate that the variable is an array. All elements of an integer array are concatenated together to create the request value. The shared integer arrays must have an extent value equal to the subscript value.

Unknown integer values sent as zeros. Unknown character values are sent as zero length strings.

Valid data types are:

%c[n]	character	n bytes
%i	signed integer	2 bytes
%u	unsigned integer	2 bytes
%l	signed long	4 bytes

Rs1..Rsn

The response values received. Each value contains two parts: the data type and a shared variable name. Response data types have the same form as request data types.

RETURNING TO PROGRESS

While running an interactive session (BTOS command with no options) you can return to PROGRESS by using the command "Exit Executive" for BTOS, or "Finish Executive" for CTOS.

The "PROGRESS Exit" command also returns you to PROGRESS, but first pauses and prompts you to "Press space bar to continue".

When ending a submit file use "PROGRESS Exit.", if you want to see the results of the last command run.

Use "Exit Executive" followed by "Finish Executive" (to run on both BTOS and CTOS), if you do not want to pause before returning to PROGRESS.

EXAMPLES

```
r-ctos.p

IF OPSYS = "unix" then UNIX ls.
   ELSE IF OPSYS = "msdos" then DOS dir.
   ELSE IF OPSYS = "os2" then OS2 dir.
   ELSE IF OPSYS = "vms" then VMS directory.
➤  ELSE IF OPSYS = "btos" then  CTOS
        "[sys]<sys>files.run" files.
   ELSE DISPLAY OPSYS  "is an unsupported operating system".
```

If the operating system you are using is UNIX, this procedure runs the UNIX ls command. If the operating system is VMS, then the procedure runs the VMS directory command. If you are using DOS, this procedure runs the DOS dir command. If you are using OS/2, this procedure runs the OS2 dir command. If you are using CTOS then the [sys] < sys > files.run files command is executed. Otherwise, a message is displayed stating the operating system is unsupported.

This sample program reads a Volume Home Block using CTOS requests.

```
readvh.p

/* readvh.p */
DEF NEW SHARED VARIABLE ercret   AS INTEGER.
DEF NEW SHARED VARIABLE vhb      AS INTEGER   EXTENT  128.
DEF NEW SHARED VARIABLE dev      AS CHARACTER INITIAL "Sys".
DEF VARIABLE FreeSectors         AS INTEGER.

/* request vhb */ ➤
CTOS OS-REQUEST 15 ercret 3 1 1 0 0 0 %c[12] dev %u[128]
vhb.

FreeSectors = vhb[55] + vhb[56] * 65536.
DISPLAY ercret dev
        SKIP(1)
        vhb[45]     LABEL "Free File Headers"
        SKIP(1)
        FreeSectors LABEL "Free Sectors" WITH SIDE-LABELS.
```

The following sample code sends an OpenFile request:

```
DEFINE NEW SHARED VAR ErcRet AS INT.
DEFINE NEW SHARED VAR Mode AS INTEGER INITIAL 28018. /* Mode Read*/
DEFINE NEW SHARED VAR FileName AS CHAR INITIAL "[sys]<sys>.user".
DEFINE NEW SHARED VAR Password AS CHAR INITIAL "".
DEFINE NEW SHARED VAR fh AS INT.

                                ④  ⑥
                   ①    ②    ③  ⑤  ⑦   ⑧    ⑨
CTOS OS-REQUEST 4 ErcRet 3 2 1 0 Mode 0 %c[60] FileName
     %c[12] Password %i fh.
       ⑩              ⑪  ⑫
```

1. request code for open file
2. error code returned
3. three integers of control information (6 bytes)
4. two request pbcb pairs to send
5. one response pbcb pair to be received
6. first control value (0)
7. second control value (Mode value)
8. third control value (0)
9. variable containing first request value
   first request value is a character with a maximum size of 60 bytes.
10. second request value, character, maximum size 12 bytes.
    variable containing second request value.
11. first response value is an integer.
12. variable containing first response value.

This next sample program reads and displays data from a file using CTOS Requests.

openfl.p

```
 /*openfl.p */

DEF NEW SHARED VAR ercret  AS INTEGER.       /* error value    */
DEF NEW SHARED VAR file AS CHARACTER.       /* file name      */
DEF NEW SHARED VAR passwd  AS CHARACTER INITIAL "". /* file password  */
DEF NEW SHARED VAR data AS CHARACTER.      /* Data read      */
DEF NEW SHARED VAR fh AS INTEGER.           /* file handle    */
DEF NEW SHARED VAR byteret AS INTEGER.        /* num bytes read */
DEF NEW SHARED VAR resv AS INTEGER INITIAL 0.  /* control reserved */
DEF NEW SHARED VAR mode    AS INTEGER INITIAL 28018. /* open mode read */


file = "[sys]<sys>.user".


/* open file */
CTOS OS-REQUEST 4 ercret 3 2 1 0 mode 0 %c[60] file %c[12] passwd %i fh.

IF ercret <> 0 THEN DO:
    MESSAGE "Open File Request Failed - Error: " ercret.
    BELL. PAUSE. LEAVE.
    END.
/* read from file */
CTOS OS-REQUEST 35 ercret 3 0 2 fh resv resv %c[512] data %i byteret.

DISPLAY   byteret LABEL "Number of Bytes Read"

   SKIP(1) data    FORMAT "x(80)" NO-LABEL
   WITH SIDE-LABELS NO-BOX.

 IF ercret <> 0 THEN DO:
    MESSAGE "Read File Request Failed - Error: " ercret.
    BELL. PAUSE. LEAVE.
    END.

 /* close file */
 CTOS OS-REQUEST 10 ercret 1 0 0 fh.

 IF ercret <> 0 THEN DO:
    MESSAGE "Close File Request Failed - Error: " ercret.
    BELL. PAUSE. LEAVE.
    END.
```

NOTES

- Running the above command in PROGRESS automatically switches contexts, runs the operating system file specified, and then swaps contexts back, returning to the PROGRESS procedure making the call. The procedure will continue to process on the next line.

- If you use the CTOS statement in a procedure, the procedure will compile on a UNIX, VMS, OS/2, or DOS system, and the procedure will run as long as flow of control does not pass through the BTOS/CTOS statement. You can use the OPSYS function to return the name of the operating system on which a procedure is being run. Using this function enables you to write applications that are fully transportable between any PROGRESS supported operating system even if they use the DOS, UNIX, VMS, OS2, and BTOS/CTOS statements.

- You cannot run executive intrinsic commands (commands contained in EXEC.RUN, which do not have actual run files) using the "CTOS command" statement. These commands include:

```
APPEND              PATH                RUN FILE
COPY                PLAYBACK            SCREEN SETUP
CREATE DIRECTORY    RECORD              SET PROTECTION
CREATE FILE         REMOVE DIRECTORY    STOP RECORD
DELETE              RENAME              TYPE
LIST                RUN                 VIDEO
LOGIN
```

- To use the *"ctos-command"*, *"run-file"*, or "SUBMIT" options you must have the Context Manager installed.

- You can also access the interactive CTOS executive by selecting the option "c" from the main menu of PROGRESS Help.

SEE ALSO OPSYS Function, UNIX Statement, VMS Statement, DOS Statement, BTOS Statement, OS2 Statement.

DATE Function

Converts three integer values representing a month, day, and year, into a date.

SYNTAX

```
DATE( month,day,year )
```

month

An expression (a constant, field name, variable name or any combination of these) whose value is an integer from 1 to 12.

day

An expression whose value is an integer from 1 to the highest valid day of the month.

year

An expression whose value is the year (e.g. 1990).

EXAMPLE

r-date.p

```
DEFINE VARIABLE cnum AS CHAR FORMAT "x(3)".
DEFINE VARIABLE cdate AS CHAR FORMAT "x(6)".
DEFINE VARIABLE iday AS INTEGER.
DEFINE VARIABLE imon AS INTEGER.
DEFINE VARIABLE iyr AS INTEGER.
DEFINE VARIABLE ddate AS DATE.

INPUT FROM VALUE(SEARCH("r-date.dat")).
REPEAT:
  SET cnum cdate.
  imon = INTEGER(SUBSTR(cdate,1,2)).
  iday = INTEGER(SUBSTR(cdate,3,2)).
  iyr = INTEGER(SUBSTR(cdate,5,2)) + 1900.
➔ ddate = DATE(imon,iday,iyr).
  DISPLAY ddate.
END.
INPUT CLOSE.
```

This procedure reads data from an input file containing date information from another system stored as character strings without slashes or dashes between month, day, and year. It converts these dates to PROGRESS dates. Note that this procedure adds 1900 to the two-digit year before using the DATE function.

SEE ALSO DAY Function, MONTH Function, WEEKDAY Function, YEAR Function

DAY Function

Converts a date to a day of the month integer value from 1 to 31.

SYNTAX

```
DAY(date)
```

date

An expression (a constant, field name, variable name or any combination of these) whose value is a date.

EXAMPLE

r-day.p

```
DEFINE VARIABLE d1 AS DATE LABEL "Date".
DEFINE VARIABLE d2 AS DATE LABEL "Same date next year".
DEFINE VARIABLE d-day AS INTEGER.
DEFINE VARIABLE d-mon AS INTEGER.

REPEAT:
  SET d1.
  d-day = DAY(d1).
  d-mon = MONTH(d1).
  IF d-mon = 2 AND d-day = 29
  THEN d-day = 28.
  d2 = DATE(d-mon,d-day,YEAR(d1) + 1).
  DISPLAY d2.
END.
```

This procedure determines the date one year from a given date, allowing for leap years. You could simply determine a date 365 days later by adding 365 to the d1 variable, but that may not produce the correct result (e.g. 1/1/84 + 365 days is 12/31/84).

SEE ALSO MONTH Function, WEEKDAY Function, YEAR Function

DBNAME Function

Returns the name of the physical database currently in use or the name of your first connected database.

SYNTAX

```
DBNAME
```

EXAMPLE

r-dbname.p

```
DEFINE VARIABLE pageno AS INTEGER FORMAT "zzz9" INITIAL 1.

FORM HEADER "Date:" TO 10 TODAY
            "Page:" AT 65 pageno SKIP
➤           "Database:" TO 10 DBNAME SKIP
            "Userid:" TO 10 USERID WITH NO-BOX NO-LABELS.
VIEW.
```

This portion of a procedure defines a header frame to hold a date, page number, database name, and userid.

NOTES

- PROGRESS returns the database name in the same form you used when you started PROGRESS. If you used a fully qualified path name to start PROGRESS, PROGRESS returns the full directory path name (such as /usr/acctg/gl on UNIX; \acctg\gl on DOS and OS/2; usr:[acctg]gl on VMS); [SYS] < acctg > gl on BTOS/CTOS. If you used a name relative to your working directory, then PROGRESS returns that name (for example, gl).

- Unless you define a format, the database name is displayed in a character field with the default format of "x(8)".

SEE ALSO Format Phrase

DBRESTRICTIONS Function

The DBRESTRICTIONS function returns a character string that describes features that are not supported for this database.

SYNTAX

$$\text{DBRESTRICTIONS} \left\{ \begin{array}{l} (\textit{integer expression}\ ) \\ (\textit{logical-name}\ ) \\ (\textit{alias}\ ) \end{array} \right\}$$

integer expression

If the parameter given to DBRESTRICTIONS is an integer expression, and there are, for example, three connected databases, then DBRESTRICTIONS(1), DBRESTRICTIONS(2), and DBRESTRICTIONS(3) return their database restrictions. Also, continuing the same example of three connected databases, DBRESTRICTIONS(4), DBRESTRICTIONS(5), etc., returns the ? value.

logical-name or *alias*

These forms of the DBRESTRICTIONS function require a quoted character string or a character expression as a parameter. An unquoted character string is not permitted. If the parameter is an alias or the logical name of a connected database, then the database restrictions string is returned. Otherwise, the ? value is returned.

EXAMPLE

dbrest.p

```
DEFINE VARIABLE i AS INTEGER.
REPEAT i = 1 TO NUM-DBS:
   DISPLAY LDBNAME(i) DBRESTRICTIONS(i) FORMAT "x(40)".
END.
```

This procedure displays the logical name and database restrictions of all connected databases.

NOTE

- DBRESTRICTIONS returns a string. This string is a comma separated list of key words representing features. Sample keywords are LAST, PREV, RECID, SETUSRID, and USE-INDEX. Each of these keywords represent a feature that is present in some PROGRESS Gateways and not in others. For example, PREV represents the feature FIND PREV which is available in PROGRESS but is not available in some other database managers.

- For example, if the database is accessed through a manager that does not support FIND PREV, then the DBRESTRICTIONS function returns the string "LAST, PREV".

- The currently supported databases and their values for the DBRESTRICTIONS function are: PROGRESS, Oracle, RMS.

- The form of the returned string makes it easy to use with the ENTRY and LOOKUP function.

- If you connect to a database with the -RO (Read-Only) parameter, the character string "Read-Only" is listed in the restrictions list for that database.

SEE ALSO CONNECT, DISCONNECT, CREATE ALIAS, and DELETE ALIAS statements; CONNECTED, LDBNAME, PDBNAME, DBTYPE, GATEWAYS, FRAME-DB, and NUM-DBS functions.

DBTYPE Function

The DBTYPE function returns the database type of a currently connected database ("PROGRESS," "RMS," "ORACLE," etc.). DBTYPE accepts as a parameter either an integer expression or a character expression.

SYNTAX

$$\text{DBTYPE} \left\{ \begin{array}{l} (\textit{integer expression}\ ) \\ (\textit{logical-name}\ ) \\ (\textit{alias}\ ) \end{array} \right\}$$

integer expression

If the parameter supplied to DBTYPE is an integer expression, and there are, for example, three currently connected databases, then DBTYPE(1), DBTYPE(2), and DBTYPE(3) return their database types. Also, continuing the same example of three connected databases, DBTYPE(4), DBTYPE(5), etc., return the ? value.

logical-name or *alias*

These forms of the DBTYPE function require a quoted character string or a character expression as a parameter. An unquoted character string is not permitted. If the parameter is an alias of a connected database or the logical name of a connected database, then the database type is returned. Otherwise, the ? value is returned.

EXAMPLE

```
r-dbtype.p

DEFINE VARIABLE i AS INTEGER.
REPEAT i = 1 TO NUM-DBS:
   DISPLAY LDBNAME(i) DBTYPE(i) FORMAT "x(40)".
END.
```

This procedure displays the logical name and database type of all connected databases.

SEE ALSO CONNECT, DISCONNECT, CREATE ALIAS, and DELETE ALIAS statements; CONNECTED, LDBNAME, PDBNAME, DBRESTRICTIONS, GATEWAYS, FRAME-DB, and NUM-DBS functions.

DBVERSION Function

DBVERSION returns a "5" if a connected database is a Version 5 database and a "6" if Version 6. For non-PROGRESS databases, you see the appropriate version of your non-PROGRESS databases. DBVERSION accepts an integer or character expression as a parameter.

SYNTAX

$$\text{DBVERSION}\left\{\begin{array}{l}(\textit{integer expression})\\(\textit{logical-name})\\(\textit{alias})\end{array}\right\}$$

integer-expression

If the parameter supplied to DBVERSION is an integer expression, and there are, for example, three currently connected databases, then DBVERSION(1), DBVERSION(2), and DBVERSION(3) return their version numbers. Continuing the same example of three connected databases, DBVERSION(4), DBVERSION (5), etc., return the ? value.

logical-name or *alias*

These forms of the DBVERSION function require a quoted character string or a character expression as a parameter. If the parameter is an alias of a connected database or the logical name of a connected database, then the version number is returned. Otherwise, the ? value is returned.

EXAMPLE

```
r-dbvers.p
```

```
DEFINE VARIABLE i AS INTEGER.
REPEAT i=1 TO NUM-DBS:
   DISPLAY DBVERSION(i) WITH 1 DOWN.
END.
```

This procedure displays the version number of all connected databases.

NOTE

- For PROGRESS databases, DBVERSION returns a "5" or a "6". For ORACLE databases, DBVERSION returns "5" or "6". For RMS and RDB databases, DBVERSION returns "RMS" and "RDB", respectively.

SEE ALSO CONNECT, DISCONNECT, CREATE ALIAS, and DELETE ALIAS statements; CONNECTED, LDBNAME, DBTYPE, PDBNAME, DBRESTRICTIONS, GATEWAYS, FRAME-DB, and NUM-DBS functions.

DECIMAL Function

Converts an expression of any data type to a decimal value.

SYNTAX

```
DECIMAL(expression)
```

expression

An expression (constant, field name, variable name or combination of these). If *expression* is character, then it must be valid for conversion into a number (e.g. "1.67" is valid, "1.x3" is not). If *expression* is logical, then the result is 0 if *expression* is false and 1 if *expression* is true. If *expression* is a date, then the result is the number of days from 1/1/4713 B.C. to that date. If the value of *expression* is unknown value (?), then the result is also unknown.

EXAMPLE

r-decml.p

```
DEFINE VARIABLE new-max AS CHARACTER FORMAT "x(10)".
REPEAT:
  PROMPT-FOR customer.cust-num WITH FRAME credit.
  FIND customer USING cust-num.
  DISPLAY cust-num name max-credit WITH FRAME credit DOWN.
  DISPLAY "Enter one of:" SKIP(1)
          "a = 5000" SKIP
          "b = 2000" SKIP
          "RETURN = 1000"
          "A dollar value" WITH FRAME vals COLUMN 60.
  SET new-max = "" WITH FRAME credit.
  IF new-max = "a"
  THEN max-credit = 5000.
  ELSE IF new-max = "b"
  THEN max-credit = 2000.
  ELSE IF new-max > "0" AND new-max < "999,999.99"
➤ THEN max-credit = DECIMAL(new-max).
  ELSE max-credit = 1000.
  DISPLAY max-credit WITH FRAME credit.
END.
```

This procedure lets the user enter new values for max-credit in a special form. If the user enters the letter "a," the procedure uses the standard "a" credit of 5000; if the user enters "b," the procedure uses a value of 2000; if the user presses RETURN, the procedure uses a value of 1000. Otherwise, the user can enter any value for max-credit. The DECIMAL function converts the value entered into a decimal value.

SEE ALSO INTEGER Function, STRING Function

DECLARE CURSOR Statement (SQL)

Associates a cursor name with a SELECT statement.

SYNTAX

```
DECLARE cursor-name CURSOR FOR SELECT-statement
```

cursor-name
The name given to the cursor

SELECT-statement
Specifies a retrieval set of rows that will be accessible when the cursor is opened.

EXAMPLE

```
DECLARE c01 CURSOR FOR SELECT * FROM
  customer WHERE cust-num < 5.
```

NOTE

- The DECLARE CURSOR statement can be used in both interactive SQL and embedded SQL.

DEFINE BUFFER Statement

PROGRESS provides you with one buffer for each file that you use in a procedure. PROGRESS uses that buffer to store one record at a time from the file as the records are needed during the procedure. If you need more than one record at a time from a file, you can use the DEFINE BUFFER statement to define additional buffers for that file. If you need to share buffers among procedures, use the DEFINE SHARED BUFFER statement.

SYNTAX

```
DEFINE [ [ NEW ] SHARED ] BUFFER buffer FOR file [ PRESELECT ]
```

NEW SHARED

Defines a buffer that can be used by other procedures. When the procedure using this statement ends, the buffer is no longer available.

SHARED

Defines a buffer that was created in another procedure with the DEFINE NEW SHARED BUFFER statement.

BUFFER *buffer*

The name of the buffer you want to define to hold records from *file*.

file

The name of the file for which you are defining an additional buffer.

To define a buffer for a file defined for multiple databases, you may need to qualify the filename with the database name. See the description of the Record Phrase for more information.

PRESELECT

If you use the PRESELECT option with a DO or REPEAT block, PROGRESS creates an internal list of the records selected. The PRESELECT option tells PROGRESS to apply that internal list to the buffer you define. You can also use the PRESELECT option in the DEFINE SHARED BUFFER statement.

EXAMPLES

The r–defb.p procedure lists every item that has a designated substitute (based on the field subs-item which is the item number of the substitute) with the number and description of the substitute. This action requires accessing two item records at once: the item buffer and the alt–item buffer.

r-defb.p

```
➤ DEFINE BUFFER alt-item FOR item.

  FOR EACH item:
    IF subs-item <> 0
    THEN DO:
      FIND alt-item WHERE
             alt-item.item-num = item.subs-item.
      DISPLAY item.item-num item.idesc alt-item.item-num
                LABEL "Alternate" alt-item.idesc.
    END.
  END.
```

The three procedures that follow gather together a group of records and then do work on those records. Specifically, the user can enter any file name and any set of record selection criteria and then look at the records in the file that meet those criteria.

r-defb2.p

```
➤ DEFINE VARIABLE fname AS CHARACTER
          FORMAT "x(12)" LABEL "Filename".
  DEFINE VARIABLE conditions AS CHARACTER
          FORMAT "x(60)" LABEL "Conditions".

  REPEAT:
    /* Get the name of a file and, optionally,
       some record selection criteria */
    UPDATE fname COLON 12 conditions COLON 12
           WITH SIDE-LABELS 1 DOWN.
    HIDE ALL.
    IF conditions <> ""
    /* Pass the filename and the record selection
       criteria as parameters */
    THEN RUN r-defb3.p fname "WHERE" conditions.
    ELSE RUN r-defb3.p fname.
  END.
```

The r-defb2.p procedure gets the name of a file (like customer) and a condition (like max-credit > 4000) and passes these as arguments to the r-defb3.p procedure.

```
                                                    r-defb3.p
➤ DEFINE NEW SHARED BUFFER rec FOR {1} PRESELECT.
  DEFINE VARIABLE flist AS CHARACTER EXTENT 12.
  DEFINE VARIABLE i AS INTEGER.
/* Look in _file the file named in the filename variable */
  FIND _file "{1}".
/* Store the file's field names in the flist array */
  FOR EACH _field OF _file USE-INDEX _field-posit:
    IF i >= 12
    THEN LEAVE.
    i = i + 1.
    flist[i] = _field._field-name.
  END.
/* Preselect records */
  DO PRESELECT EACH rec {2} {3} {4} {5} {6} {7}
                        {8} {9} {10} {11} {12}:
/* Pass the filename and all field names to r-defb4.p */
  RUN r-defb4.p "{1}" flist[1] flist[2] flist[3]
                      flist[4] flist[5] flist[6]
                      flist[7] flist[8] flist[9]
                      flist[10] flist[11] flist[12].
  END.
```

The r-defb3.p procedure does several things:

- The PROGRESS Data Dictionary contains a file called _file. That file contains a single record for each of your database files. If you go into the Dictionary, you can look at the structure of _file.

- Suppose the user supplies "customer" as a filename. The FIND statement in the r-defb3.p procedure translates to "FIND _file "customer". The FIND statement finds, in _file, the record for the customer file.

- The PROGRESS Data Dictionary also contains a file called _field. That file contains a single record for each of your database fields. The FOR EACH statement reads the name of each of those fields into the flist array variable. Assuming a filename of customer, the flist array variable contains the names of each of the fields in the customer file.

- Assuming that the condition the user supplied with the filename is "max-credit > 4000", the DO PRESELECT EACH rec statement translates to "DO PRESELECT EACH rec WHERE max-credit > 4000". PROGRESS goes through the customer file and selects the records that meet the criteria. It creates a temporary file containing a pointer to each of those records. This list of preselected records is associated with the rec buffer.

- Runs r-defb4.p, passing the filename (customer) and the names of all of the fields in that file.

The r-defb4.p procedure has access to the rec buffer (and through it to the set of preselected records). This connection is made by using PRESELECT on the DEFINE SHARED BUFFER statement. The r defb4.p procedure displays those records.

r-defb4.p

```
DEFINE SHARED BUFFER rec FOR {1} PRESELECT.

REPEAT:
  FIND NEXT rec.
  DISPLAY {2} {3} {4} {5} {6} {7} {8} {9} {10}
          {11} {12} {13} WITH 1 COLUMN 1 DOWN.
END.
```

Because rdefb3.p and r-defb4.p use run-time argument passing, they cannot be precompiled. By having separate versions of r-defb4.p for each file, and running the appropriate one in r-defb3.p, response time should be improved. This approach would be worthwhile if there were many lines of code in r-defb4.p.

NOTES

- Every statement that uses a file name to refer to the default buffer can also use the name of a defined alternate buffer.

- All data definitions and field names are associated with a file, not a buffer. Data definitions and field names remain the same no matter what buffer you use.

- If two buffers contain the same record, a change to one of the buffers is automatically reflected in the other buffer.

- If you define a NEW SHARED BUFFER in a procedure, call a subprocedure that puts a record into that buffer, and display the buffer in the main procedure, PROGRESS displays a "Missing FOR, FIND or CREATE for file" message.

For example:

```
/* Main procedure */

DEFINE NEW SHARED BUFFER x FOR customer.
RUN proc2.p.
DISPLAY x.
```

```
/* proc2.p */

DEFINE SHARED BUFFER x FOR customer.
FIND FIRST x.
```

When you run or compile the main procedure, PROGRESS displays the message "Missing FOR, FIND, or CREATE for customer" because the FIND statement is not in the main procedure. To avoid this problem, explicitly scope the customer record to the main procedure block. For example:

```
/* Main procedure */

DEFINE NEW SHARED BUFFER x FOR customer.
RUN proc2.p.
DO FOR x:
  DISPLAY x.
END.
```

DEFINE PARAMETER Statement

Defines a run-time parameter in a called subprocedure. Each parameter requires its own DEFINE statement. The parameters must be specified in the RUN statement in the same order they are defined with DEFINE statements. In addition, the parameter types (INPUT, OUTPUT, and INPUT-OUTPUT) specified in the DEFINE and RUN statements must agree. The data types must agree as well.

SYNTAX

```
DEFINE { INPUT        } PARAMETER parameter
       { OUTPUT       }
       { INPUT-OUTPUT }

{ AS datatype }  [ COLUMN-LABEL label [ ! label ]...  ] ...
{ LIKE field  }  [ DECIMALS n                         ]
                 [ FORMAT string                      ]
                 [ INITIAL { [ constant ,... ] }      ]
                 [ LABEL string                       ]
                 [ NO-UNDO                            ]
```

INPUT PARAMETER
: Defines a parameter that will be passed a value by a RUN statement in a called procedure.

OUTPUT PARAMETER
: Defines a parameter that will return a value to the called procedure.

INPUT-OUTPUT PARAMETER
: Defines a parameter that will receive a value from, and return a value to, the called procedure.

parameter
: The name of the parameter being defined.

See the DEFINE VARIABLE statement for descriptions of the AS, LIKE, COLUMN-LABEL, DECIMALS, FORMAT, INITIAL, LABEL, and NO-UNDO options.

EXAMPLES

In the example below, the r-runpar.p procedure runs a subprocedure called r-param.p and passes the subprocedure an INPUT parameter. The subprocedure r-param.p displays the INPUT parameter.

```
                                                    r-runpar.p
➤RUN r-param.p (INPUT 10).
```

```
                                                    r-param.p
➤DEFINE INPUT PARAMETER int-param AS INTEGER.
 DISPLAY int-param LABEL "Integer input param"
             WITH SIDE-LABELS.
```

In the example below, the r-runprl.p procedure runs a subprocedure called r-paraml.p. This example illustrates the use of multiple parameters and shows that the parameters must be passed in the proper order and must be of the same datatype. Note that if you don't specify a parameter type in the run statement, then PROGRESS assumes it is an input parameter.

```
                                                         r-runprl.p
DEFINE VARIABLE new-param AS CHARACTER FORMAT "x(20)".
DEFINE VARIABLE out-param AS DECIMAL.
DEFINE VARIABLE in-param AS INTEGER INIT 20.

RUN r-paraml.p (OUTPUT out-param, 10, OUTPUT new-param,
                                               in-param).
DISPLAY out-param LABEL "Updated YTD Sales" SKIP
                  new-param LABEL "Status" WITH SIDE-LABELS.
```

r-param1.p

```
DEFINE OUTPUT PARAMETER xout-param AS DECIMAL.
DEFINE INPUT PARAMETER newin AS INTEGER.
DEFINE OUTPUT PARAMETER xnew-param AS CHARACTER.
DEFINE INPUT PARAMETER xin-param AS INTEGER.

FOR EACH customer:
   xout-param = ytd-sls + xout-param.
END.
DISPLAY xout-param LABEL "YTD Sales" WITH SIDE-LABELS.
xout-param = xout-param + newin + xin-param.
xnew-param = "Example Complete".
```

In the example below, the `r-runpr2.p` procedure displays information from a database file and assigns the value of a database field to a variable called `io-param`. The variable is passed as an INPUT-OUTPUT parameter to a subprocedure called `r-param2.p`. The subprocedure `r-param2.p` performs a calculation on the INPUT-OUTPUT parameter and then passes it back to the main procedure. The `r-runpr2.p` assigns the value `io-param` to a database field and then displays `io-param`.

r-runpr2.p

```
DEFINE VARIABLE io-param AS INTEGER.
FOR EACH item:
        DISPLAY Idesc On-hand WITH 1DOWN.
        io-param = Item.On-hand.
      → RUN r-param2.p (INPUT-OUTPUT io-param).
        Item.On-hand = io-param.
        DISPLAY io-param LABEL "New Quantity On-hand".
END.
```

r-param2.p

```
→DEFINE INPUT-OUTPUT PARAMETER io-param AS INTEGER.
 DEFINE VARIABLE inp-qty AS INTEGER.
 PROMPT-FOR inp-qty LABEL "Quantity Received?".
 ASSIGN inp-qty.
 io-param = io-param + inp-qty.
```

SEE ALSO DEFINE VARIABLE statement, RUN statement.

DEFINE SHARED FRAME Statement

Defines a frame for use within a procedure or within several procedures.

SYNTAX

```
DEFINE [ NEW ] SHARED FRAME frame
```

NEW SHARED FRAME *frame*
: Defines a frame to be shared by a procedure called directly or indirectly by the current procedure. The called procedure must name the same frame in a DEFINE SHARED FRAME statement.

SHARED FRAME *frame*
: Defines a frame that was created by another procedure that used the DEFINE NEW SHARED FRAME statement. When you use the DEFINE SHARED FRAME statement, you may not name any fields or variables in that frame that are not already named in the frame described by the DEFINE NEW SHARED FRAME statement.

EXAMPLE

r-shrfrm.p

```
➤DEFINE NEW SHARED FRAME cust-frame.
 DEFINE NEW SHARED VARIABLE csz
        AS CHARACTER FORMAT "x(22)".
 DEFINE NEW SHARED BUFFER xcust FOR customer.

 FOR EACH xcust WHERE cust-num <= 20:

   {r-shrfrm.i}  /* include file for layout of
                    shared frame */

   DISPLAY name phone address sales-rep
           city + ", " + st + " " + STRING(zip) @ csz
           max-credit WITH FRAME cust-frame.

   RUN r-updord.p. /* External procedure to update
                      customer's orders */
 END.
```

This procedure defines the shared frame cust-frame. It also defines a shared variable and a shared buffer. For each customer whose customer number is less than 20, the procedure displays some customer information in the cust-frame. The format for the cust-frame is defined in the r-shrfrm.i include file.

r-shrfrm.i

```
FORM
  xcust.name        COLON 10   xcust.phone        COLON 50
  xcust.address     COLON 10   xcust.sales-rep    COLON 50
  csz NO-LABEL      COLON 12   xcust.max-credit   COLON 50
  SKIP(2)
  order.order-num   COLON 10   order.odate        COLON 30
  order.sdate       COLON 30
  order.pdate       COLON 30

  WITH SIDE-LABELS 1 DOWN CENTERED ROW 5
    TITLE "Customer/Order Form" FRAME cust-frame.
```

After the r-shrfrm.p procedure displays some customer information, it calls the r-updord.p procedure.

r-updord.p

```
➤ DEFINE SHARED FRAME cust-frame.
  DEFINE SHARED VARIABLE csz
         AS CHARACTER FORMAT "x(22)".
  DEFINE SHARED BUFFER xcust FOR customer.

  FOR EACH order OF xcust:

    {r-shrfrm.i}  /* include file for layout of
                     shared frame */

    DISPLAY order.order-num WITH FRAME cust-frame.
    UPDATE order.odate order.sdate order.pdate
           WITH FRAME cust-frame.
  END.
```

The r-updord.p procedure defines the variable, frame, and buffer that were originally defined in the r-shrfrm.p procedure. However, in this second reference to the items, the keyword NEW is omitted. The r-updord.p procedure displays, and allows you to update, the order information for the customer displayed in the cust-frame. The order information is displayed in this same frame.

NOTES

- You can use just one DEFINE SHARED FRAME statement per frame in a procedure.
- If you name variables in a shared frame, PROGRESS does not automatically make those variables shared. If you want to share the variables between procedures, you must use the DEFINE SHARED VARIABLE statement.
- PROGRESS scopes a new shared frame to the block in which you first reference that frame. (The DEFINE FRAME statement does not count as a reference.) PROGRESS scopes a shared frame outside of the called procedure.
- The frame-phrase options specified in a DEFINE NEW SHARED FRAME statement are carried over to all corresponding DEFINE SHARED FRAME statements and cannot be overridden.
- You can use different field level help and validation in new shared and shared frames.
- You must define a shared frame before referencing that frame in a procedure.
- All frame fields in a shared frame must first be defined in a FORM statement. If you use aggregate phrases to accumulate values within shared frames, you must use the ACCUM option in each procedure that uses the shared frame. See the FORM statement for more information.

SEE ALSO DEFINE BUFFER Statement, DEFINE VARIABLE Statement, FORM Statement

DEFINE STREAM Statement

PROGRESS automatically provides two unnamed streams to each procedure: the input stream and the output stream. These streams give the procedure a way to communicate with an input source and an output destination. For example, the following statement tells PROGRESS to use the unnamed input stream to get input from the file named `testfile`.

```
INPUT FROM testfile.
```

You use the DEFINE STREAM statement when you want to use streams other than the two unnamed streams supplied by PROGRESS. Using other streams lets you, in a single procedure, get input from more than one source simultaneously or send output to more than one destination simultaneously.

SYNTAX

```
DEFINE [ [ NEW        ] SHARED ] STREAM stream
         [ NEW GLOBAL ]
```

NEW SHARED STREAM *stream*

Defines a stream that can be shared by other procedures. When the procedure using the DEFINE NEW SHARED STREAM statement ends, the stream is no longer available to any procedure. The value you use for the *stream* option must be a constant.

NEW GLOBAL SHARED STREAM *stream*

Defines a stream that can be shared by other procedures and that will remain available even after the procedure containing the DEFINE NEW GLOBAL SHARED STREAM statement ends. The value you use for the *stream* option must be a constant.

SHARED STREAM *stream*

Defines a stream that was created by another procedure using the DEFINE NEW SHARED STREAM statement or the DEFINE NEW GLOBAL SHARED STREAM statement. The value you use for the *stream* option must be a constant.

STREAM *stream*

Defines a stream that can be used only by the procedure containing the DEFINE STREAM statement. The value you use for the *stream* option must be a constant.

EXAMPLE

```
                                                    r-dfstr.p
DEFINE NEW SHARED STREAM rpt.
DEFINE STREAM exceptions.
DEFINE VARIABLE fnr AS CHARACTER FORMAT "x(12)".
DEFINE VARIABLE fne AS CHARACTER FORMAT "x(12)".
DEFINE VARIABLE excount AS INTEGER
  LABEL "Total number of exceptions".
DEFINE NEW SHARED BUFFER xitem FOR item.

SET fnr LABEL "Enter filename for report output" SKIP(1)
    fne LABEL "Enter filename for exception output"
    WITH SIDE-LABELS FRAME fnames.

OUTPUT STREAM rpt TO VALUE(fnr) PAGED.
OUTPUT STREAM exceptions TO VALUE(fne) PAGED.
FOR EACH xitem:
  IF on-hand < alloc THEN DO:
    DISPLAY STREAM exceptions item-num idesc
      on-hand alloc WITH FRAME exitem DOWN.
    excount = excount + 1.
  END.
  RUN r-dfstr2.p.
END.
DISPLAY STREAM exceptions SKIP(1) excount
        WITH FRAME exc SIDE-LABELS.
DISPLAY STREAM rpt WITH FRAME exc.

OUTPUT STREAM rpt CLOSE.
OUTPUT STREAM exceptions CLOSE.
```

```
                                                    r-dfstr2.p
DEFINE SHARED STREAM rpt.
DEFINE SHARED BUFFER xitem FOR item.

DISPLAY STREAM rpt item-num idesc WITH NO-LABELS NO-BOX.
```

This procedure, in a single pass through the item file, uses the rpt stream to create a report and the exceptions stream to create a list of exceptions. (The DISPLAY statement in the r-dfstr2.p procedure should be put in the r-dfstr.p procedure for efficiency. It is in a separate procedure here to illustrate shared streams.)

NOTES

- Using the DEFINE STREAM statement creates a stream, but it does not actually open that stream. To open a stream, you must use the STREAM option with the INPUT FROM, INPUT THROUGH, OUTPUT TO, OUTPUT THROUGH, or INPUT-OUTPUT THROUGH statements. You must also use the STREAM option with any data handling statements that move data to and from the stream.
- After you open the stream, you can use the SEEK function to return the offset value of the file pointer, or you can use the SEEK statement to position the file pointer to any location in the file.

SEE ALSO DISPLAY Statement, INPUT CLOSE Statement, INPUT FROM Statement, INPUT THROUGH Statement, INPUT-OUTPUT THROUGH Statement, OUTPUT CLOSE Statement, OUTPUT THROUGH Statement, OUTPUT TO Statement, PROMPT-FOR Statement, SEEK Function, SEEK Statement, SET Statement, Chapter 9 of the *Programming Handbook*

DEFINE VARIABLE Statement

Defines a variable (a temporary field) for use within a procedure or within several procedures.

SYNTAX

```
DEFINE [ [ NEW            ] SHARED ] VARIABLE variable
         [ NEW GLOBAL     ]

                      [ [NOT] CASE-SENSITIVE                      ]
                      [ COLUMN-LABEL label [ ! label ] ...        ]
   { AS datatype }    [ DECIMALS n                                ]
   { LIKE field  }    [ EXTENT   n                                ] ...
                      [ FORMAT string                             ]
                      [ INITIAL { [constant ,...] }               ]
                      [ LABEL string                              ]
                      [ NO-UNDO                                   ]
```

NEW SHARED VARIABLE

Defines a variable to be shared by a procedure called directly or indirectly by the current procedure. The called procedure must name the same variable in a DEFINE SHARED VARIABLE statement.

NEW GLOBAL SHARED VARIABLE

Defines a variable that can be used by any procedures that name that variable using the DEFINE SHARED VARIABLE statement. The value of a global shared variable remains available throughout a PROGRESS session.

SHARED VARIABLE

Defines a variable that was created by another procedure that used the DEFINE NEW SHARED VARIABLE or DEFINE NEW GLOBAL SHARED VARIABLE statement.

VARIABLE

Defines a variable whose value is available only within the current procedure.

variable

The name of the variable you are defining.

AS *data type*

Indicates the data type of the variable you are defining. The data types are CHARACTER, DATE, DECIMAL, INTEGER, LOGICAL, and RECID.

LIKE *field*

Indicates the name of the variable or database field whose characteristics you want to use for the variable you are defining. If you name a variable with this option, you must have defined that variable earlier in the procedure. You can override the format, label, initial value, decimals, and extent of the variable or database field by using the FORMAT, LABEL, INITIAL, DECIMALS, and EXTENT options. If you do not use these options, the variable takes on the characteristics of the variable or database field you name.

If *field* has help and validate options defined, the variable you are defining does not take on those characteristics.

The LIKE option in a DEFINE VARIABLE statement, DEFINE WORKFILE statement, or format phrase requires that a particular database be connected. Since you can startup a PROGRESS application session without connecting to a database, you should use the LIKE option with caution.

[NOT] CASE-SENSITIVE

Indicates that the value stored for a character variable is case-sensitive, and that all comparisons operations involving the variable are case-sensitive. If you do not use this option, PROGRESS comparisons will normally be case-insensitive. If you define a variable LIKE another field of variable, the new variable inherits is case-sensitivity. You can use [NOT] CASE-SENSITIVE to override this default.

COLUMN-LABEL *label* [!*label*]...

Names the label you want to display above the variable data in a frame that uses column labels. If you want the label to use more than one line (you want the label to be "stacked"), use a separate *label* for each line and separate each *label* with a exclamation point (!). For example:

r-collbl.p

```
DEFINE VARIABLE credit-percent AS INTEGER
➤ COLUMN-LABEL "Enter   !percentage!increase ".

FOR EACH customer:
  DISPLAY name max-credit.
  SET credit-percent.
  max-credit = (max-credit * (credit-percent / 100))
               + max-credit.
  DISPLAY max-credit @ new-credit LIKE max-credit
    LABEL "New max cred".
END.
```

PROGRESS does not display column labels if you use the SIDE-LABELS or NO-LABELS options with the frame phrase.

If you define a variable to be LIKE a field, and that field has been defined in the Dictionary to have a column label, the variable inherits that column label.

If you want to use the exclamation point (!) character as one of the characters in a column label, you must use two exclamation point characters ("!!").

DECIMALS *n*

When you define a variable AS DECIMAL, PROGRESS automatically stores up to 10 decimal places for the value of that variable. You use the DECIMALS option to store a smaller number of decimal places. The DECIMALS option has nothing to do with the display format of the variable, just the storage format.

If you use the LIKE option to name a field whose definition you want to use to define a variable, PROGRESS uses the number of decimals in the field definition to determine how many decimal places to store for the variable.

EXTENT *n*

The extent of an array variable. If you are using the AS *datatype* option and you do not use the EXTENT option (or specify *n* as zero), the variable is not an array variable. If you are using the LIKE *field* option and you do not use the EXTENT option, the variable uses the extent defined for the database field you name.

If you want to define a variable that is like an array variable or field but you do not want the variable to be an array, you can use EXTENT 0 to indicate a non-array field.

FORMAT *string*

The data format of the variable you define. If you use the AS *datatype* option and you don't use FORMAT *string*, the variable uses the default format for its data type. Table 7 lists the default data formats for the six of the data types (you cannot display a raw datatype):

Table 7: Default Data Type Display Formats

Data Type	Default Display Format
Character	x(8)
Date	99/99/99
Decimal	->>,>>9.99
Integer	->,>>>,>>9
Logical	yes/no
Recid	>>>>>>9

See Chapter 4 of the *Programming Handbook* for more information about data types and data formatting.

If you are using the LIKE *field* option and you do not use the FORMAT *string* option, the variable uses the format defined for the database field you name. *string* must be enclosed in quotes.

INITIAL {[*constant*,...]}

The initial value of the variable you are defining. If you are using the AS *datatype* option and you do not use the INITIAL *constant* option, the default is the initial value for the data type of the variable.

If you are defining an array variable, you can supply initial values for each element in the array. For example:

```
DEFINE VARIABLE array-var
  AS CHARACTER EXTENT 3 INITIAL ["Add","Delete","Update"].
```

If you do not supply enough values to fill up the elements of the array, PROGRESS puts the last value you named into the remaining elements of the array. If you supply too many values, you receive an error message. Table 8 lists the default initial values for the six variable data types:

Table 8: Default Variable Initial Values

Data Type of Variable	Default Initial Value
Character	Blank
Date	? (Unknown value - displays as blanks for dates)
Decimal	0
Integer	0
Logical	no
Raw	a zero length sequence of bytes
Recid	?

If you are using the LIKE *field* option and you do not use the INITIAL *constant* option, the variable uses the initial value of the field or variable. In the DEFINE SHARED VARIABLE statement, the INITIAL option has no effect. However, the DEFINE NEW SHARED VARIABLE and the DEFINE NEW WORKFILE statements do work with the INITIAL option.

LABEL *string*

The label you want to use when the variable is displayed. If you are using the AS *datatype* option and you do not use the LABEL *string* option, the default label is the variable name. If you are using the LIKE *field* option and you do not use the LABEL *string* option, the variable uses the label of the field or variable you name. *string* must be enclosed in quotes.

NO-UNDO

When the value of a variable is changed during a transaction and the transaction is undone, the value of the variable is restored to its prior value. If you do not want, or if you do not need, the value of a variable to be undone even when it has been changed during a transaction, use the NO-UNDO option with the DEFINE VARIABLE statement. NO-UNDO variables are often much more efficient, and you should carefully consider when you can do without the default undo service.

Specifying NO-UNDO for a variable is especially useful if you want to indicate an error condition as the value of the variable, perform an UNDO, and later take some action based on that error condition. If one variable is defined LIKE another that is NO-UNDO, the second variable will be NO-UNDO only if you explicitly specify NO-UNDO in the definition of that second variable.

EXAMPLES

r-dfvar.p

```
➤ DEFINE NEW SHARED VARIABLE del AS LOGICAL.
➤ DEFINE NEW SHARED VARIABLE nrecs AS INTEGER.
  del = no.
  MESSAGE "Do you want to delete the orders
           being printed (y/n)?"
  UPDATE del.
  RUN r-dfvar2.p.
  IF del
  THEN MESSAGE nres
       "Orders have been shipped and were deleted".
  ELSE MESSAGE nrecs "Orders have been shipped".
```

```
                                                 /* r-dfvar2.p */
➤ DEFINE SHARED VARIABLE del AS LOGICAL.
  DEFINE SHARED VARIABLE nrecs AS INTEGER.

  OUTPUT TO PRINTER.
  nrecs = 0.
  FOR EACH order WHERE sdate <> ?:
    nrecs = nrecs + 1.
    FOR EACH order-line OF order:
      DISPLAY order-num line-num qty price.
      IF del THEN DELETE order-line.
    END.
    IF del THEN DELETE order.
  END.
  OUTPUT CLOSE.
```

The r-dfvar.p procedure defines two variables, del and nrecs to be shared with procedure r-dfvar2.p. The del variable is used to pass information to r-dfvar2.p, while nrecs serves to pass information back to r-dfvar.p from r-dfvar2.p.

r-dfvar3.p

```
➤ DEFINE NEW GLOBAL SHARED VARIABLE first-time
         AS LOGICAL INITIAL TRUE.
➤ DEFINE VARIABLE selection AS INTEGER FORMAT "9"
         LABEL "Selection".

  IF first-time
  THEN DO:
    RUN r-init.p.
    first-time = false.
  END.
  FORM
  "    MAIN MENU          " SKIP(1)
  "1 - Accounts Payable   " SKIP
  "2 - Accounts Receivable" WITH CENTERED ROW 5 FRAME menu.
  REPEAT:
    VIEW FRAME menu.
    UPDATE selection AUTO-RETURN WITH FRAME sel
      CENTERED ROW 12 SIDE-LABELS.
    IF selection = 1 THEN DO:
      HIDE FRAME menu.
      HIDE FRAME sel.
      RUN apmneu.p.
    END.
    ELSE IF selection = 2 THEN DO:
      HIDE FRAME menu.
      HIDE FRAME sel.
      RUN armenu.p.
    END.
    ELSE DO:
      MESSAGE "Invalid selection. Try again".
      UNDO, RETRY.
    END.
  END.
```

This partial procedure is intended as a startup procedure. It defines a new global variable with the initial value true and uses that variable to determine whether or not to run an initialization procedure, r-init.p, that displays sign-on messages. Then the global variable first-time is set to false. If this procedure were restarted during the same session (e.g. the user pressed STOP), r-init.p would not run again.

The procedure also defines the variable selection to be used solely within the procedure for entering menu choices.

r-dfvar4.p

```
DEFINE VARIABLE dow AS CHARACTER FORMAT "x(9)" EXTENT 7
       INITIAL ["Sunday", "Monday", "Tuesday", "Wednesday",
                "Thursday", "Friday", "Saturday"].

DEFINE VARIABLE dob AS DATE INITIAL TODAY.

REPEAT WITH SIDE-LABELS 1 DOWN CENTERED ROW 10
       TITLE "Date of Birth":
  DISPLAY SKIP(1).
  UPDATE dob LABEL "Enter date of birth".
  DISPLAY dow[WEEKDAY(dob)] LABEL "It was a".
END.
```

This procedure finds the day of the week of a date the user enters. The procedure defines an array with seven elements and uses the INITIAL option to define the intial value of each element in the array.

NOTES

- You can use the DEFINE VARIABLE statement anywhere in a procedure. However, all references to the variable must appear after the DEFINE VARIABLE statement that defines it.
- You should use the CASE-SENSITIVE option only when it is important to distinguish between uppercase and lowercase values entered for a character variable. For example, you would need to use CASE-SENSITIVE to define a variable for a part number that contains mixed uppercase and lowercase characters.
- After you use the DEFINE NEW GLOBAL SHARED VARIABLE statement to create a global shared variable, you use DEFINE SHARED VARIABLE statements in other procedures to access that variable.
- You cannot create a global variable twice in the same PROGRESS session. If you try and the definitions of the two variables do not match, you receive an error. If the definitions of the two variables do match, PROGRESS disregards the second variable you tried to create (on the assumption that you are re-running a startup procedure).
- Changes made to variables when there is no active transaction are not undone when a block is undone.
- When a procedure names a shared variable, PROGRESS does the following if the procedure actually uses the variable:

  - PROGRESS searches through the calling chain of procedures looking for the most recent DEFINE NEW SHARED VARIABLE statement that created that shared variable.
  - If no DEFINE NEW SHARED VARIABLE statement is found, PROGRESS searches for a DEFINE NEW GLOBAL SHARED VARIABLE statement that created the shared variable.
  - Once PROGRESS finds one of these statements it does not search any further for other statements that may have defined the same variable as NEW or GLOBAL.

- PROGRESS checks the definition of a SHARED variable against that of the corresponding NEW SHARED or NEW GLOBAL SHARED variable. The data types and array extents must match. If the FORMAT, LABEL and DECIMALS specifications are not the same, each procedure uses its individual specification. The DEFINE NEW statement determines whether a shared variable is NO-UNDO.

SEE ALSO DEFINE BUFFER Statement

DEFINE WORKFILE Statement

Defines a work file (a temporary database file) for use within a procedure or within several procedures.

SYNTAX

```
DEFINE [ [ NEW ] SHARED ] WORKFILE  workfile-name [ NO-UNDO ]

    [ LIKE file-name  ]

    [ FIELD field-name        [ [NOT]CASE-SENSITIVE                   ]     ]
    [                         [ COLUMN-LABEL label [ !label ] ...     ]     ]
    [    { AS datatype }      [ DECIMALS  n                           ]     ]
    [    { LIKE field  }      [ EXTENT n                              ] ... ] ...
    [                         [ FORMAT  string                        ]     ]
    [                         [ INITIAL { [constant,...] }            ]     ]
    [                         [ LABEL  string                         ]     ]
```

NEW SHARED WORKFILE
: Defines a work file to be shared by a procedure called directly or indirectly by the current procedure. The called procedure must name the same work file in a DEFINE SHARED WORKFILE statement.

SHARED WORKFILE
: Defines a work file that was defined by another procedure that used the DEFINE NEW SHARED WORKFILE statement.

WORKFILE
: Defines a work file whose records are available only within the current procedure.

workfile-name
: The name of the work file you are defining.

NO-UNDO
: When a work file record is changed during a transaction and the transaction is undone, the record is restored to its prior condition. If you do not want, or if you do not need, the work file record to be undone even when it has been changed during a transaction, use the NO-UNDO option with the DEFINE WORKFILE statement. NO-UNDO work files are very efficient, and you should carefully consider when you can do without the default undo service.

LIKE *file-name*

Indicates the name of a file whose characteristics you want to use for the work file you are defining. All of the fields of this "base" file will also be in the work file.

To define a work file like another file that is defined for multiple databases, you must qualify the filename with the database name. See the description of the Record Phrase for more information.

FIELD *field-name*

The name of a field in the work file.

AS *datatype*

Indicates the data type of the field or variable you are defining. The data types are CHARACTER, DATE, DECIMAL, INTEGER, LOGICAL, and RECID.

LIKE *field*

Indicates the name of the variable or database field whose characteristics you want to use for the work file field or variable you are defining. If you name a variable with this option, you must have defined that variable earlier in the procedure. You can override the format, label, initial value, decimals, and extent of the variable or database field by using the FORMAT, LABEL, INITIAL, DECIMALS, and EXTENT options. If you do not use these options, the field or variable takes on the characteristics of the variable or database field you name.

The LIKE option in a DEFINE VARIABLE statement, DEFINE WORKFILE statement, or format phrase requires that a particular database be connected. Since you can startup a PROGRESS application session without connecting to a database, you should use the LIKE option with caution.

CASE-SENSITIVE

Indicates that the value stored for a character field is case-sensitive, and that all comparisons made to it are also case-sensitive. If you do not use this option, PROGRESS comparisons will be case-insensitive.

COLUMN-LABEL *label* [!*label*] ...

Names the label you want to display above the variable data in a frame that uses column labels. If you want the label to use more than one line (you want the label to be "stacked"), use a separate *label* for each line and separate each *label* with a exclamation point (!). For example:

```
r-collbl.p

DEFINE VARIABLE credit-percent AS INTEGER
  COLUMN-LABEL "Enter   !percentage!increase ".

FOR EACH customer:
  DISPLAY name max-credit.
  SET credit-percent.
  max-credit = (max-credit * (credit-percent / 100))
               + max-credit.
  DISPLAY max-credit @ new-credit LIKE max-credit
    LABEL "New max cred".
END.
```

PROGRESS does not display column labels if you use the SIDE-LABELS frame-phrase.

If you define a variable to be LIKE a field, and that field has been defined in the Dictionary to have a column label, the variable inherits that column label.

If you want to use the exclamation point (!) character as one of the characters in a column label, you must use two exclamation point characters ("!!").

DECIMALS *n*

When you define a field as DECIMAL, PROGRESS automatically stores up to 10 decimal places for the value of that field. You use the DECIMALS option to store a smaller number of decimal places. The DECIMALS option has nothing to do with the display format of the field, just the storage format.

If you use the LIKE option to name a field whose definition you want to use to define a work file field, PROGRESS uses the number of decimals in the field definition to determine how many decimal places to store for the field.

EXTENT *n*

The extent of an array field. If you are using the AS *datatype* option and you do not use the EXTENT option (or specify *n* as zero), the field is not an array field. If you are using the LIKE *field* option and you do not use the EXTENT option, the field uses the extent defined for the database field you name.

If you want to define a field that is like an array field but you do not want the work file field to be an array, you can use EXTENT 0 to indicate a non-array field.

FORMAT *string*

The data format of the field you are defining. If you are using the AS *datatype* option and you do not use the FORMAT *string* option, the field uses the default format for its data type. Table 9 lists the default data formats for the six data types:

Table 9: Default Data Type Display Formats

Data Type	Default Display Format
Character	x(8)
Date	99/99/99
Decimal	->>,>>9.99
Integer	->,>>>,>>9
Logical	yes/no
Recid	>>>>>>9

See Chapter 4 in the *Programming Handbook* for more information about data types and data formatting.

If you are using the LIKE *field* option and you do not use the FORMAT *string* option, the field uses the format defined for the database field you name. *string* must be enclosed in quotes.

INITIAL {[*constant*,...]}

The initial value of the variable you are defining. If you are using the AS *datatype* option and you do not use the INITIAL *constant* option, the default is the initial value for the datatype of the variable.

If you are defining an array variable, you can supply initial values for each element in the array. For example:

```
DEFINE VARIABLE array-var
  AS CHARACTER EXTENT 3 INITIAL ["Add","Delete","Update"].
```

If you do not supply enough values to fill up the elements of the array, PROGRESS puts the last value you name into the remaining elements of the array. If you supply too many values, you receive an error message. Table 10 lists the default initial values for the six variable data types.

Table 10: Default Variable Initial Values

Data Type of Variable	Default Initial Value
Character	Blank
Date	? (Unknown value - displays as blanks for dates)
Decimal	0
Integer	0
Logical	no
Recid	?

If you are using the LIKE *field* option and you do not use the INITIAL *constant* option, the variable uses the initial value of the field or variable. In the DEFINE SHARED VARIABLE statement, the INITIAL option has no effect.

LABEL *string*

The label you want to use when the field is displayed. If you are using the AS *datatype* option and you do not use the LABEL *string* option, the default label is the field name. If you are using the LIKE *field* option and you do not use the LABEL *string* option, the field uses the label of the field or variable you name. You must enclose *string* in quotes.

EXAMPLE

r-wrkfil.p

```
DEFINE WORKFILE showsales
  FIELD region    LIKE customer.sales-region LABEL "REGION"
  FIELD state     LIKE customer.st LABEL "STATE"
  FIELD tot-sales LIKE customer.ytd-sls COLUMN-LABEL
                       "TOTAL!SALES".

FOR EACH customer BREAK BY customer.st:
  ACCUMULATE ytd-sls (TOTAL BY customer.st).
  IF LAST-OF(customer.st)
  THEN DO:
    CREATE showsales.
    showsales.state = customer.st.
    showsales.tot-sales = ACCUM TOTAL BY customer.st
                                        ytd-sls.
    showsales.region = customer.sales-region.
  END.
END.

FOR EACH showsales BREAK BY region BY state:
  IF FIRST-OF(region)
  THEN DISPLAY region.
  DISPLAY state tot-sales (TOTAL BY region).
END.
```

The r-wrkfil.p procedure accumulates all year-to-date sales by state and stores that information for later display. The procedure uses work files to accomplish this task.

The r-wrkfil.p procedure defines the work file showsales. The work file contains the three fields named region, state, and tot-sales. These fields have all the same characteristics (except labels) as the customer.sales-region, customer.st, and customer.ytd-sls fields, respectively.

The first FOR EACH loop in the the r-wrkfil.p procedure sorts customers by state. Then it accumulates the year-to-date sales for each customer by state. When the procedure finds the last customer in a state, it creates a showsales file for that state. The procedure assigns information to the fields in the showsales file. After looking at each customer, the procedure continues to the next FOR EACH statement.

The second FOR EACH statement in the r-wrkfil.p procedure uses the information stored in the showsales file. Because you treat a work file within a procedure the same way you treat a database file, you can perform the same work with the showsales file that you can with a database file.

NOTES

- When finding records in a work file, you must use either FIRST, LAST, NEXT, or PREV with the FIND statement, unless you are finding a record using its recid.
- PROGRESS disregards the following when used in conjunction with a work file:
  - The VALIDATE option on a DELETE statement.
  - The SHARE-LOCK, EXCLUSIVE-LOCK, and NO-LOCK options used with the FIND or FOR statements.
  - The NO-WAIT option on the FIND statement.
- When you use the AMBIGUOUS function in conjunction with a work file, the function always returns a value of FALSE.
- Complete work file definitions must be included in a DEFINE SHARED WORKFILE statement and shared work files must be defined identically.
- These are the differences between work files and regular database files:
  - PROGRESS does not use the PROGRESS database manager (and server for multi-user systems) when working with work files.
  - Work files do not have indexes.
  - If you do not explicitly delete the records in a work file, PROGRESS discards those records, and the work file, at the end of the procedure which initially defined the work file.
  - Users do not have access to each other's work files.
- Because you cannot index a work file, PROGRESS uses the following rules for storing records in a work file:
  - If you create a series of work file records without doing any other record operations, PROGRESS orders the newly created records in chronological order.
  - If you use the FIND PREV statement when at the beginning of a work file and then create a work file record, PROGRESS stores that record at the beginning of the work file.

  - When you use the FIND statement to find a work file record and then use the CREATE statement to create a new work file record, PROGRESS stores that new record after the record you just found.

- Data handling statements that cause PROGRESS to automatically start a transaction for a regular file will not cause PROGRESS to automatically start a transaction for a work file. If you want to start a transaction for operations involving a work file, you must explicitly start a transaction by using the TRANSACTION keyword.

- Work files are private:

  - Even if two users define work files with the same name, the work files are private: one user cannot see records the other user has created.

  - If two procedures run by the same user define work files with the same name, PROGRESS treats those work files as two separate files unless the SHARED option is included in both procedures.

- DEFINE SHARED WORKFILE does not automatically provide a shared buffer. If you want to use a shared buffer with a shared work file, you must define that buffer.

- Work file records are built in 64-byte "chunks." Approximately the first 60 bytes of the first chunk in each record are taken up by record specification information, or a "record header." That is, if a record is 14 bytes long, it will be stored in two- 64 byte chunks, using the first 60 bytes as a record header. If the record is 80 bytes long, it will fit into three 64-byte chunks. The first chunk contains 60 bytes of header information plus the first 4 bytes of the record; the second chunk contains 64 bytes of the record; and the last chunk contains the remaining record bytes.

- The NO-UNDO option in a work file definition overrides a transaction UNDO for CREATE, UPDATE, DELETE, and RELEASE statements accessing the work file, regardless of whether these statements are executed before or during the transaction block that is undone.

- A transaction UNDO overrides a FIND statement accessing a work file defined with the NO-UNDO option, regardless of whether the find is executed before or during the transaction that is undone.

- You should use the CASE-SENSITIVE option only when it is important to distinguish between uppercase and lowercase values entered for a character field. For example, you would need to use CASE-SENSITIVE to define a field for a part number that contains mixed uppercase and lowercase characters.

- You can use the Data Dictionary to generate a DEFINE WORKFILE statement based on the fields of an existing file. Enter the Data Dictionary. Press **U** (Utilities), **G** (Generate Include Files), and **W** (DEFINE WORKFILE statement). Select a file with the arrow keys or by typing its first few letters, and then press RETURN. Specify a name for the include file; it is named *file-name*.i by default. For example, below is an include file that includes a DEFINE WORKFILE statement for the fields in the file `order-line` from the PROGRESS demo database. Now you can use the PROGRESS editor to add or delete one or more fields or to make other adjustments.

```
order-ln.i

/* 10/10/89 workfile definition for file order-line */
/* {1} = "", "NEW" or "NEW SHARED"  */
/* {2} = "" or "NO-UNDO" */

DEFINE {1} WORKFILE worder-line {2} /* LIKE order-line */
  FIELD Disc      AS INTEGER FORMAT ">>9" LABEL "Disc %"
  FIELD Item-num  AS INTEGER FORMAT "99999" LABEL "Item num"
  FIELD Line-num  AS INTEGER FORMAT ">>9" LABEL "Line num"
  FIELD Order-num AS INTEGER FORMAT ">>>>9" LABEL "Order num"
  FIELD Price     AS DECIMAL DECIMALS 2
                     FORMAT "->,>>>,>>9.99" LABEL "Price"
  FIELD Qty       AS INTEGER FORMAT "->>>>9" LABEL "Qty"
  FIELD Qty-ship  AS INTEGER LABEL "Qty ship".
```

Now, the following line, inserted in any procedure, defines a workfile based on the `order-line` file.

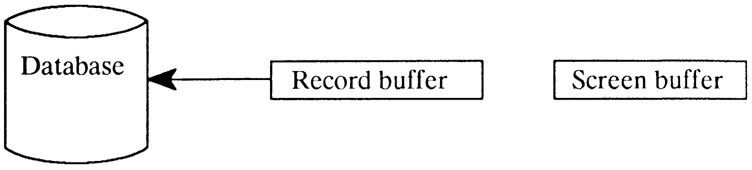
```
{order-ln.i  " "  "NO-UNDO"}
```

SEE ALSO CREATE Statement, DEFINE BUFFER Statement, FIND Statement, Format Phrase, { } Argument Reference, { } Include File.

DELETE Statement

Removes a record from a record buffer and from the database.

DATA MOVEMENT



SYNTAX

```
DELETE record   [ VALIDATE ( condition, msg-expression  ) ]
```

record

The name of a record buffer. You can delete a record only after it has been put into a record buffer by a CREATE, FIND, FOR EACH, or INSERT statement.

If you define an alternate buffer for a file, you can delete a record from that buffer by using the name of the buffer with the DELETE statement.

To delete a record in a file defined for multiple databases, you must qualify the record's filename with the database name. See the description of the Record Phrase for more information.

VALIDATE (*condition,msg-expression*)

You use the VALIDATE option to specify a logical value that allows the deletion of a record when TRUE, but does not allow the deletion of a record when FALSE.

condition is a boolean expression (a constant, field name, variable name, or any combination of these) whose value is either TRUE or FALSE.

msg-expression is the message you want to display if the *condition* is FALSE. You must enclose *msg-expression* in quotation marks ("").

You can also describe delete validation criteria for a file in the Dictionary. To suppress the Dictionary delete validation criteria for a file, use the following VALIDATE option:

```
VALIDATE(TRUE,"")
```

If you use the DELETE statement to delete a record in a work file, PROGRESS disregards any VALIDATE option you use with the DELETE statement.

EXAMPLES

```
r-delet.p

FOR EACH customer:
  DELETE customer.
END.
```

The r-delet.p procedure deletes all the records in the customer file.

```
r-delet2.p

DEFINE VARIABLE del AS LOGICAL FORMAT "y/n".
REPEAT:
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num.
  DISPLAY name.
  del = no.
  UPDATE del LABEL
    "Enter ""y"" to confirm delete".
  IF del THEN DELETE customer.
END.
```

The r-delet2.p procedure prompts the user for a customer number and then displays the name of that customer. It then asks the user to press "y" to confirm deletion of the customer record. The user's response is stored in the del variable. If the value of the del variable is "y," the procedure deletes the customer record.

r-delval.p

```
DEFINE VARIABLE ans AS LOGICAL.

REPEAT WITH 1 DOWN:
  PROMPT-FOR customer.cust-num.
  FIND customer USING customer.cust-num.
  DISPLAY name.
  ans = no.
  DISPLAY "Do you want to delete this customer ?"
           WITH FRAME f-query.
  UPDATE ans WITH FRAME f-query NO-LABELS.
  IF ans
  THEN DELETE customer
       VALIDATE(NOT(CAN-FIND(order OF customer)),
       "This customer has outstanding orders and cannot
       be deleted.").
END.
```

The r-delval.p procedure prompts the user for a customer number. The procedure displays the name of the customer and asks the user, "Do you want to delete this customer ?" If the user answers no, the procedure prompts the user for another customer number. If the user answers yes, the procedure checks whether the customer has orders, using the VALIDATE option. If they do have orders, the procedure displays this message, "This customer has outstanding orders and cannot be deleted." If the customer has no orders, the procedure deletes the customer.

NOTES

- When you run procedures that delete large numbers of records (e.g. a month end file purge), the process may run much faster if you use the -i or /INTEGRITY start-up option in single-user mode. (You must back up your database before using this option). See Chapter 3 of *System Administration II: General* for more information on start-up options.

- Deleting records does not change the amount of space the database takes up on the disk. PROGRESS re-uses RECIDs. It does not delete the RECID when a record is deleted. To recover disk space, you need to dump and reload your database.

SEE ALSO CREATE Statement, FIND Statement, FOR EACH Statement, INSERT Statement

DELETE ALIAS Statement

The DELETE ALIAS Statement deletes an alias from the alias table.

SYNTAX

```
DELETE ALIAS alias.
```

alias
A previously existing alias. It can be an unquoted string, a quoted string, or VALUE (*expression*).

EXAMPLE

```
r-dalias.p

DELETE ALIAS myalias.
```

This procedure deletes the alias "`myalias`" from the alias table.

NOTES

- If a precompiled program needs an alias and you delete the alias, then the program will not run.
- If an attempt is made to delete a nonexistent alias, nothing happens.

SEE ALSO CONNECT, DISCONNECT, and CREATE ALIAS statements; CONNECTED, LDBNAME, PDBNAME, DBTYPE, DBRESTRICTIONS, GATEWAYS, FRAME-DB, and NUM-DBS functions.

DELETE FROM Statement (SQL)

Deletes one or more rows from a table.

SYNTAX

```
DELETE FROM table-name [ WHERE { search-condition |
  { CURRENT OF cursor-name}}]
```

FROM *table-name*
  The name of the table from which you want to delete rows.

WHERE *search-condition*
  Identifies the rows to be deleted. If you omit the WHERE clause, all rows of the target table will be deleted.

WHERE CURRENT OF cursor-name
  Identifies the cursor that points to the row to be deleted (positioned delete).

EXAMPLE

```
DELETE FROM customer
   WHERE cust-num > 100.
```

NOTE

- The DELETE FROM statement can be used in both interactive SQL and embedded SQL.

SEE ALSO Chapter 15 *Programming Handbook*

DICTIONARY Statement

Runs the PROGRESS Data Dictionary.

SYNTAX

```
DICTIONARY
```

EXAMPLE

```
r-dict.p

DEFINE VARIABLE ans AS LOGICAL.

DISPLAY "Do you want to access the Dictionary?"
        WITH ROW 7 COLUMN 20 NO-LABELS.
UPDATE ans.
IF ans THEN DICTIONARY.
```

This procedure runs the Dictionary if the user answers "yes" to a prompt.

NOTES

- The DICTIONARY statement is equivalent to "RUN dict.p" — it runs the PROGRESS procedure called "dict.p". PROGRESS uses the regular search rules to find the dictionary procedure. The dictionary procedure is part of the PROGRESS system software.
- You can also access the Dictionary from the main menu of PROGRESS Help.
- PROGRESS Query/Run-Time provides a restricted version of the Dictionary.

SEE ALSO Chapter 3 of the *PROGRESS Language Tutorial*

DISCONNECT Statement

The DISCONNECT statement disconnects the specified database.

SYNTAX

```
DISCONNECT logical-name.
```

logical name

A logical database name. It may be an unquoted string, a quoted string, or VALUE (*expression*). The *logical-name* is previously set, at start-up or with a connect statement, by using the `-ld` *logical-name* option. If a logical name was not specified using the `-ld` option, then the physical database filename, minus the .db suffix, is the default logical name.

EXAMPLE

r-discnt.p

```
DISCONNECT mydb.
```

This procedure disconnects the database with logical name `mydb`.

NOTES

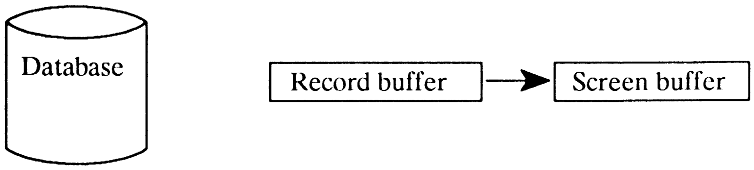
- If a transaction is active for *logical name*, DISCONNECT is deferred until the transaction completes or is undone. If a CONNECT statement for the same *logical name* database is executed before the same transaction completes or is undone, then the pending CONNECT and DISCONNECT cancel each other and the database remains connected.
- If there are any active procedures that refer to the database, then the disconnect will not take effect until they all complete execution.
- When the database referred to by *logical-name* is disconnected, existing aliases for *logical-name* remain in existence. Later, if you connect to a database with the same *logical-name*, the same alias is still available.

SEE ALSO CONNECT, CREATE ALIAS, and DELETE ALIAS statements; CONNECTED, LDBNAME, PDBNAME, DBTYPE, DBRESTRICTIONS, GATEWAYS, FRAME-DB, and NUM-DBS functions.

DISPLAY Statement

Moves data to a screen buffer, displaying the data on the screen or other output destination. PROGRESS uses frames to display data. A frame describes how constant and variable data is arranged for display and data entry. You can either let PROGRESS construct default frames or you can explicitly describe frames and their characteristics.

DATA MOVEMENT



SYNTAX

```
DISPLAY [ STREAM stream ]
      [ expression  [ format-phrase ]                                 ]
      [             [ WHEN expression ]                               ]
      [             [ ( aggregate-phrase ) ]                          ]
      [                                                               ] ...
      [ expression  [ FORMAT string   ] ... @base-field               ]
      [             [ WHEN expression ]                               ]
      [             [ format-phrase ]                                 ]
      [                                                               ]
      [  SPACE [( n )]                                                ]
      [  SKIP[( n )]                                                  ]
      [ frame-phrase ]
```

```
DISPLAY [ STREAM stream ]  record  [EXCEPT field ...] [ frame-phrase ]
```

STREAM *stream*

Specifies the name of a stream. If you do not name a stream, the unnamed stream is used. See the DEFINE STREAM reference page and Chapter 9 of the *Programming Handbook* for more information about streams.

expression

A constant, field name, variable name, or any combination of these that results in the value you want to display.

If *expression* is a simple field or variable, PROGRESS checks to see if that particular field or variable has been used previously in the same frame. If it has, PROGRESS displays the field or variable in the same frame field as the earlier instance of that field or variable.

In the case of array fields, array elements with constant subscripts are treated just like any other field. Array fields with no subscripts are expanded as though you had typed in the implicit elements.

If you reference a[i] in the same frame in which you reference a or a[constant], a[i] overlays the appropriate frame field based on the value of i, rather than being displayed in a new frame field for a[i]. For example:

r-array.p

```
1  DEFINE VARIABLE i AS INTEGER.
2  FOR EACH item:
3    DISPLAY item-num idesc mnth-ship.
4    DO i = 1 TO 12:
5      SET mnth-shp[i] WITH 1 COLUMN.
6    END.
7    DISPLAY mnth-shp
8      WITH FRAME aaa COLUMN 40 1 COLUMN.
9  END.
```

Here, mnth-shp[i] is referenced in the same frame in which mnth-shp is referenced. That is, line 5 references mnth-shp[i] and line 3 references mnth-shp. Both references use the same frame. Therefore, instead of creating a new frame field for mnth-shp[i], PROGRESS uses the same frame fields as are created for the entire mnth-shp array.

r-array.2.p

```
 1  DEFINE VARIABLE i AS INTEGER.
 2  FOR EACH item:
 3    DISPLAY item-num idesc
 4            mnth-shp[1] mnth-shp[2].
 5    DO i = 1 TO 12:
 6      SET mnth-shp[i].
 7    END.
 8    DISPLAY mnth-shp
 9      WITH FRAME aaa COLUMN 40 1 COLUMN.
10  END.
```

Here, line 4 references only elements 1 and 2. Therefore, when PROGRESS tries to overlay mnth-shp[i] in line 6, there is only room for elements 1 and 2. You get an error after entering data for those 2 elements. The following example shows a solution to that problem:

r-array.3.p

```
 1  DEFINE VARIABLE i AS INTEGER.
 2  FOR EACH item:
 3    DISPLAY item-num idesc
 4            mnth-shp[1] mnth-shp[2] WITH 6 DOWN.
 5    FORM i mnth-shp[i].
 6    DO i = 1 TO 12:
 7      DISPLAY i.
 8      SET mnth-shp[i].
 9    END.
10    DISPLAY mnth-shp
11      WITH FRAME aaa 2 COLUMNS.
12  END.
```

If you explicitly reference a[i] in a FORM statement, regular array fields (mnth-shp[1] and mnth-shp[2] in this example) are not overlaid.

format-phrase

Specifies one or more frame attributes for a field, variable, or expression.

Here is the syntax for *format-phrase*:

```
[ AT n                                           ]
[ AS  datatype                                   ]
[ ATTR-SPACE                                     ]
[ AUTO-RETURN                                    ]
[ BLANK                                          ]
[ COLON n                                        ]
[ COLUMN-LABEL label [ ! label  ]...             ]
[ FORMAT string                                  ] ...
[ HELP string                                    ]
[ LABEL string                                   ]
[ LIKE field                                     ]
[ NO-ATTR-SPACE                                  ]
[ NO-LABEL                                       ]
[ TO n                                           ]
[ VALIDATE (condition, msg-expression)           ]
```

For more information on format-phrase, see the FORMAT Phrase reference page.

WHEN *expression*

Displays an item only when the expression used in the WHEN option has a value of TRUE. Here, *expression* is a field name, variable name, or any combination of these whose value is logical.

aggregate-phrase

Identifies one or more aggregate values to be calculated optionally based on a change in a break group. Here is the syntax for *aggregate-phrase*:

```
{ AVERAGE       }
{ COUNT         }
{ MAXIMUM       }
{ MINIMUM       }
{ TOTAL         }
{ SUB-AVERAGE   } ...  [ BY break-group  ] ...
{ SUB-COUNT     }
{ SUB-MAXIMUM   }
{ SUB-MINIMUM   }
{ SUB-TOTAL     }
```

For more information on *aggregate-phrase*, see the AGGREGATE Phrase reference page.

FORMAT *string*

The format in which you want to display the *expression*. If you do not use the FORMAT option, PROGRESS uses the defaults shown in Tables 11 and 12.

Table 11: Default Display Formats

Type of Expression	Default Format
Field	Format from Dictionary
Variable	Format from variable definition
Constant character	Length of character string
Other	Default format for the data type of the expression. Table 12 shows these default formats.

Table 12: Default Data Type Display Formats

Data Type of Expression	Default Format
Character	x(8)
Date	99/99/99
Decimal	->>,>>9.99
Integer	->,>>>,>>9
Logical	yes/no

base-field

The name of the field whose frame area you want to use. The *base-field* must be the name of a field or variable; it cannot be an expression or constant.

PROGRESS reserves enough space for the *base-field* to hold the longest format displayed there. All right-justified fields (numerics that do not use side labels) are right justified within the reserved area. The label is left or right justified according to the *base-field*. Whenever you enter data into the *base-field*, PROGRESS blanks out any characters to the left or right of the area used by the field being displayed.

PROGRESS underlines a screen area that is the longer of the *base-field* and the overlaying field. However, you can enter as many characters as there are spaces in the format of the field.

To determine the format to use for displaying the expression at the *base-field*, PROGRESS looks at the following and uses the first that applies:

- An explicit FORMAT phrase used with the *expression*.
- If the expression is a character string constant, a format that accommodates that string.
- If the data type of the expression matches that of the *base-field*, the format of the *base-field*.
- The standard format of the expression as if it were displayed without a *base-field*.

SPACE (*n*)
Identifies the number (*n*) of blank spaces to be inserted after the expression is displayed. *n* can be zero (0). If the number of spaces is more than the spaces left on the current line of the frame, a new line is started and extra spaces are discarded. If you do not use this option or do not use *n*, one space is inserted between items in the frame.

SKIP (*n*)
Identifies the number (*n*) of blank lines to be inserted after the expression is displayed. *n* can be zero (0). If you do not use this option, a line is not skipped between expressions unless the expressions do not fit on one line. If you use the SKIP option but do not specify *n*, or if *n* is 0, a new line is started unless it is already at the beginning of a new line.

frame-phrase
Specifies the overall layout and processing properties of a frame, with the syntax:

```
       [ ACCUM                                                  ]
       [ ATTR-SPACE                                             ]
       [ CENTERED                                               ]
       [          { [ DISPLAY ] color-phrase }                  ]
       [ COLOR    { PROMPT color-phrase      } ...              ]
       [ COLUMN  expression                                     ]
       [ n COLUMNS                                              ]
       [ DOWN                                                   ]
       [ expression DOWN                                        ]
       [ FRAME frame                                            ]
       [ NO-ATTR-SPACE                                          ]  ...
WITH   [ NO-BOX                                                 ]
       [ NO-HIDE                                                ]
       [ NO-LABELS                                              ]
       [ NO-UNDERLINE                                           ]
       [ NO-VALIDATE                                            ]
       [ OVERLAY                                                ]
       [ PAGE-BOTTOM                                            ]
       [ PAGE-TOP                                               ]
       [ RETAIN n                                               ]
       [ ROW  expression                                        ]
       [ SCROLL   n                                             ]
       [ SIDE-LABELS                                            ]
       [ TITLE [ COLOR  color-phrase   ]  expression            ]
       [ TOP-ONLY                                               ]
       [ WIDTH  n                                               ]
```

For more information on *frame-phrase*, see the Frame Phrase reference page.

record

The name of the record you want to display. Naming a record is a shorthand way of listing each field individually.

To display a record in a file defined for multiple databases, you must qualify the record's filename with the database name. See the description of the Record Phrase for more information.

EXCEPT *field*

All fields except those fields listed in the EXCEPT phrase will be displayed.

EXAMPLES

r-disp.p

```
    FOR EACH customer BY st BY name:
→     DISPLAY st cust-num name.
      FOR EACH order OF customer:
→       DISPLAY order-num name sdate pdate.
        FOR EACH order-line OF order, item OF order-line:
→         DISPLAY line-num idesc qty price.
        END.
      END.
    END.
```

This procedure generates a hierarchical report of customers (sorted by state and name), the orders belonging to those customers, and the order-lines belonging to each order.

r-disp2.p

```
  FOR EACH order, customer OF order:
➡  DISPLAY order-num customer.name sdate pdate.
    FOR EACH order-line OF order, item OF order-line:
➡    DISPLAY line-num idesc qty price qty * price (TOTAL)
            LABEL "Order-value".
    END.
  END.
```

This procedure lists each order, some customer information, and the order-lines for each order. In addition, the procedure calculates an order-value for each of the order-lines of an order, and totals those values to produce a total value for an entire order.

r-disp3.p

```
  FOR EACH customer:
➡  DISPLAY name SKIP address SKIP address2 SKIP
            city + ", " + st
            FORMAT "x(16)" zip WHEN zip NE 0
            SKIP(2) WITH NO-BOX NO-LABELS.
  END.
```

The r-disp3.p procedure displays a name and address list in a format you can use for mailing labels. The SKIP and FORMAT options are used to produce a standard address format. The WHEN option suppresses the display of the zip code field if there is no zip code value in the field (otherwise it displays 00000).

NOTES

- When PROGRESS compiles a procedure, it uses a top to bottom pass of the procedure to design all the frames needed by that procedure, adding field and related format attributes as it goes through the procedure.
- Data you are displaying on a terminal should not contain special control characters such as tabs, form feeds, or backspaces.
- If you use a single qualified identifier with the DISPLAY statement, the compiler first interprets the reference as *dbname.filename*. If the compiler cannot resolve the reference as *dbname.filename*, it tries to resolve it as *filename.fieldname*.

SEE ALSO ACCUM Function, Aggregate Phrase, DOWN Statement, EXPORT Statement, FORM Statement, Format Phrase, Frame Phrase, MESSAGE Statement, PAGE Statement, PUT Statement, PUT SCREEN Statement, UP Statement

DO Statement

Groups statements into a single block, optionally specifying various processing services or block properties.

SYNTAX

```
[ label : ] DO
   [ FOR record [ , record ] ... ]
   [ PRESELECT[ EACH ] record-phrase                                        ]
   [              [ , [ EACH ] record-phrase ] ...                          ]
   [              [ [ BREAK ] { BY expression  [ DESCENDING ] } ... ]       ]
   [ variable = expression1   TO expression2   [ BY k ] ]
   [ WHILE expression ]
   [ TRANSACTION ]
   [ ON ENDKEY-phrase ]
   [ ON ERROR-phrase ]
   [ frame-phrase ]
```

FOR *record* [, *record*]

Names the buffer you want to work with in the block and scopes the buffer to the block. The scope of a record determines when the buffer for that record is cleared and written back to the database. See Chapter 5 of the *Programming Handbook* for more information on record scoping.

To work with a record in a file defined for multiple databases, you must qualify the record's filename with the database name. See the description of the Record Phrase for more information.

PRESELECT [EACH] *record-phrase*

Goes through a file, selecting records that meet the criteria you specify in *record-phrase*. PRESELECT creates a temporary index that contains pointers to each of the preselected records in the database file. Then you can use other statements such as FIND NEXT to process those records.

If you name multiple files with PRESELECT, any sorting you specify applies to all files. If you then do a FIND on the last file in the PRESELECT list, records are read into the buffers for all of the files in the list.

The *record-phrase* option identifies the criteria to use when preselecting records. Here is the syntax for *record-phrase*:

```
record [ constant ] [ WHERE expression                  ] ...
                    [ USING field [ AND field ]...      ]
                    [ OF file                           ]
                    [ USE-INDEX index                   ]

                    [ SHARE-LOCK     ]
                    [ EXCLUSIVE-LOCK ]
                    [ NO-LOCK        ]
```

For more information about *record-phrase*, see the RECORD Phrase reference page.

If, within a PRESELECT block, you find a record using the recid of that record, PROGRESS disregards any other selection criteria you applied to the PRESELECT. For example, suppose the recid of order number 4 is 1234. In this example:

```
DO PRESELECT EACH order WHERE order-num > 5:
  FIND FIRST order WHERE RECID(order) = 1234.
  DISPLAY order.
END.
```

PROGRESS finds and displays order number 4 even though the selection criteria specifies that the order number must be greater than 5. The recid always overrides other selection criteria. Furthermore, note that if you use FIND *recid*, the index cursor is not reset in the preselected list. That is, even if record *recid* is in the preselected list, FIND NEXT does **not** find the record that follows it in the preselected list.

BREAK

Indicates that subgroups will be used for aggregation and with the FIRST, LAST, FIRST-OF and LAST-OF functions. If you use BREAK you must also use BY.

BY *expression* [DESCENDING]

Sorts the preselected records by the value of *expression*. If you do not use the BY option, PRESELECT sorts the records in order by the index used to extract the records. The DESCENDING option sorts the records in descending order as opposed to the default ascending order.

variable = *expression1* TO *expression2* [BY *k*]

The name of a field or variable whose value you are incrementing in a loop. *expression1* is the starting value for *variable* on the first iteration of the loop. *k* is the amount to add to *variable* after each iteration and must be a constant. When *variable* exceeds *expression2* (or is less than *expression2* if *k* is negative) the loop ends. Since *expression1* is compared to *expression2* at the start of the first iteration of the block, the block may be executed 0 times. *expression2* is reevaluated on each iteration of the block.

WHILE *expression*

Indicates the condition under which you want the DO block to continue processing the statements within it. Using the WHILE option turns a DO block into an iterating block, the block iterates as long as the condition specified by the expression is true. The expression is any combination of constants, field names, and variable names that yield a logical value.

TRANSACTION

Identifies the DO block as a system transaction block. PROGRESS starts a system transaction for each iteration of a transaction block if there is not already an active system transaction. See Chapters 5 and 8 of the *Programming Handbook* for more information about transactions.

ON ENDKEY-*phrase*

Describes the processing that takes place when the ENDKEY condition occurs during a block. Here is the syntax for the ON ENDKEY-*phrase:*

```
                                 [ , LEAVE  [ label2 ] ]
ON ENDKEY UNDO[ label1 ]         [ , NEXT   [ label2 ] ]
                                 [ , RETRY  [ label2 ] ]
                                 [ , RETURN            ]
```

ON ERROR-*phrase*

Describes the processing that takes place when there is an error during a block. Here is the syntax for the ON ERROR-*phrase:*

```
                                 [ , LEAVE  [ label2 ] ]
ON ERROR  UNDO[ label1 ]         [ , NEXT   [ label2 ] ]
                                 [ , RETRY  [ label2 ] ]
                                 [ , RETURN            ]
```

frame-phrase

Specifies the overall layout and processing properties of a frame.

Here is the syntax of *frame-phrase*:

```
      [ ACCUM                                                    ]
      [ ATTR-SPACE                                               ]
      [ CENTERED                                                 ]
      [         { [ DISPLAY ] color-phrase }                     ]
      [ COLOR   { PROMPT color-phrase      } ...                 ]
      [ COLUMN expression                                        ]
      [ n COLUMNS                                                ]
      [ DOWN                                                     ]
      [ expression DOWN                                          ]
      [ FRAME frame                                              ]
WITH  [ NO-ATTR-SPACE                                            ] ...
      [ NO-BOX                                                   ]
      [ NO-HIDE                                                  ]
      [ NO-LABELS                                                ]
      [ NO-UNDERLINE                                             ]
      [ NO-VALIDATE                                              ]
      [ OVERLAY                                                  ]
      [ PAGE-BOTTOM                                              ]
      [ PAGE-TOP                                                 ]
      [ RETAIN n                                                 ]
      [ ROW expression                                           ]
      [ SCROLL n                                                 ]
      [ SIDE-LABELS                                              ]
      [ TITLE [ COLOR color-phrase ] expression                  ]
      [ TOP-ONLY                                                 ]
      [ WIDTH n                                                  ]
```

For more information on *frame-phrase*, see the Frame Phrase reference page.

EXAMPLE

r-do.p

```
  FOR EACH customer:
    DISPLAY name max-credit.
    PAUSE 3.
➡   IF max-credit > 500 THEN DO:
      max-credit = 500.
      DISPLAY max-credit.
    END.
  END.
```

This procedure goes through the customer file and, for those customers whose max-credit is over 500, reduces max-credit to 500. The procedure uses an unmodified DO block to process two statements in the event that max-credit is over 500. Unmodified DO blocks are most useful in conditional, or IF...THEN...ELSE, situations.

NOTE

- Use a DO statement rather than a REPEAT statement when you loop through each element of an array. Within a transaction, this avoids separate subtransactions being created with corresponding overhead.

  For example:

```
DO i = 1 TO 12:
   mnth-sales[i] = 0.
END.
```

- is more efficient than:

```
REPEAT i = 1 TO 12:
  mnth-sales[i] = 0.
END.
```

SEE ALSO FIND Statement, FOR Statement, Frame Phrase, ON ENDKEY Phrase, ON ERROR Phrase, Record Phrase, REPEAT Statement

DOS Statement

Runs a program, DOS command, DOS batch file, or starts the DOS command processor, allowing interactive processing of DOS commands.

SYNTAX

```
DOS [ SILENT ] [ dos-command              [ argument                ] ... ]
               [ VALUE( expression )      [ VALUE( expression )     ]     ]
```

SILENT

After processing a DOS statement, PROGRESS pauses, telling you to press the space bar to continue. When you press the space bar, PROGRESS clears the screen and continues processing. You can use the SILENT option to eliminate this pause. Use this option only if you are sure that the DOS program, command, or batch file will not generate output to the screen.

dos-command

The name of the program, command, or batch file you want to run. If you do not use this option, the DOS statement starts the DOS command processor and remains at DOS level until you type "exit."

VALUE*(expression)*

expression in this option is an expression (a constant, field name, variable name, or any combination of these) whose value is the name of a program, command, or batch file you want DOS to execute.

argument

One or more arguments you want to pass to the program, command, or batch file being run by the DOS statement. These arguments are expressions that PROGRESS converts to character values if necessary.

EXAMPLE

```
r-dos.p

IF OPSYS = "UNIX" then UNIX ls.
  ELSE IF OPSYS = "MSDOS" then DOS dir.
  ELSE IF OPSYS = "OS2" then OS2 dir.
  ELSE IF OPSYS = "VMS" then VMS directory.
  ELSE IF OPSYS = "BTOS" then  BTOS
  "[sys]<sys>files.run" files.
  ELSE DISPLAY OPSYS  "is an unsupported operating system".
```

If the operating system you are using is UNIX, this procedure runs the UNIX ls command. If the operating system is VMS, then the procedure runs the VMS directory command. If you are using DOS, this procedure runs the DOS dir command. If you are using OS/2, this procedure runs the OS2 dir command. If you are using BTOS then the [sys] < sys > files.run files command is executed. Otherwise, a message is displayed stating the operating system is unsupported.

NOTES

- If, for example, you use the OS2 statement in a procedure, the procedure will compile on, for example, a UNIX system, and the procedure will run as long as flow of control does not pass through the DOS statement while running on UNIX. You can use the OPSYS function to return the name of the operating system on which a procedure is being run. Using this function enables you to write applications that are fully transportable between any PROGRESS supported operating system even if they use the DOS, UNIX, etc. statements.

- You can also access the interactive DOS command processor by selecting option "e" from the main menu of PROGRESS Help.

SEE ALSO OPSYS Function, UNIX Statement, VMS Statement, BTOS Statement, CTOS Statement, OS2 Statement

DOWN Statement

A down frame is a frame that can display multiple occurrences of the set of fields defined in the frame. Use the DOWN option on the FRAME phrase to specify a down frame. If you do not use the DOWN option, PROGRESS automatically makes certain frames down frames, unless you specify otherwise.

On the first iteration of a block, PROGRESS displays the first set of data (a record, field, or variable value) in the first occurrence of a frame. After displaying the data, PROGRESS advances to the next occurrence in the frame on the second iteration of the block, and displays the second set of data there. For more information on frames and down frames, see Chapter 7 of the *Programming Handbook*.

When the block to which a frame belongs iterates, PROGRESS automatically advances one frame line. Use the DOWN statement if you want to explicitly move to a different display line at any time.

SYNTAX

DOWN [STREAM *stream*] [*expression*] [*frame-phrase*]

STREAM *stream*

Specifies the name of a stream. If you do not name a stream, PROGRESS uses the unnamed stream. See the DEFINE STREAM reference page and Chapter 9 of the *Programming Handbook* for more information about streams.

expression

The number of occurrences of data in the frame that you want to move down.

DOWN is the same as DOWN 1, except for the following:

1. Nothing happens until the next data handling statement that affects the screen.
2. Several DOWN statements in a row with no intervening displays are treated like a single DOWN 1.

DOWN 0 does nothing. If *n* is negative, the result is the same as UP n.

frame-phrase

Specifies the overall layout and processing properties of a frame.

Here is the syntax of *frame-phrase*:

```
     ┌ ACCUM                                                ┐
     │ ATTR-SPACE                                           │
     │ CENTERED                                             │
     │        ┌ [ DISPLAY ] color-phrase ┐                   │
     │ COLOR  └ PROMPT color-phrase      ┘ ...              │
     │ COLUMN expression                                    │
     │ n COLUMNS                                            │
     │ DOWN                                                 │
     │ expression DOWN                                      │
     │ FRAME frame                                          │
WITH │ NO-ATTR-SPACE                                        │
     │ NO-BOX                                               │ ...
     │ NO-HIDE                                              │
     │ NO-LABELS                                            │
     │ NO-UNDERLINE                                         │
     │ NO-VALIDATE                                          │
     │ OVERLAY                                              │
     │ PAGE-BOTTOM                                          │
     │ PAGE-TOP                                             │
     │ RETAIN n                                             │
     │ ROW expression                                       │
     │ SCROLL n                                             │
     │ SIDE-LABELS                                          │
     │ TITLE [ COLOR color-phrase ] expression              │
     │ TOP-ONLY                                             │
     └ WIDTH n                                              ┘
```

EXAMPLE

r-down.p

```
DEFINE VARIABLE laststate AS CHARACTER.

FOR EACH customer BY state:
  IF st < > laststate THEN DO:
    IF laststate < > "" THEN DOWN 1.
    laststate = st.
  END.
  DISPLAY cust-num name address city st.
END.
```

This procedure prints a customer report, sorted by state, skipping one line after the last customer in each state.

NOTES

- After displaying a down frame, PROGRESS automatically advances to the next frame line on each iteration of the block to which the frame belongs. This is true whether or not you use the DOWN statement. If you do not want PROGRESS to do this automatic advancing, name the frame outside of the block involved (the statement FORM WITH FRAME *frame* is enough to name a particular frame and cause that frame to be scoped to the higher block).

- When PROGRESS reaches the last frame line and encounters a DOWN statement, it clears the frame and starts at the top line of the frame, unless you used the SCROLL option on the frame. In that case, PROGRESS scrolls the frame up one iteration only, to make room for the next iteration.

SEE ALSO DEFINE STREAM Statement, Frame Phrase, UP Statement, SCROLL Statement, Chapter 7 of the *Programming Handbook*

DROP INDEX Statement (SQL)

Removes an index.

SYNTAX

```
DROP INDEX index-name
```

index-name
The name of the index you want to remove.

EXAMPLE

```
DROP INDEX item_num.
```

NOTES

- The DROP INDEX statement can be used only in interactive SQL.
- Only the owner of an index can remove that index.

DROP TABLE Statement (SQL)

Removes a table from the database. It also removes all indexes defined on that table and all access privileges, as well as all data.

SYNTAX

```
DROP TABLE table-name
```

table-name
The name of the table you want to remove.

EXAMPLE

```
DROP TABLE employee.
```

NOTES

- The DROP TABLE statement can be used only in interactive SQL.
- Only the owner of a table can remove that table.
- PROGRESS/SQL does not allow you to delete a table that is referenced in a view definition.

DROP VIEW Statement (SQL)

Removes a view from the database.

SYNTAX

```
DROP VIEW view-name
```

view-name
The name of the view you want to remove.

EXAMPLE

```
DROP VIEW doc.
```

NOTES

- The DROP VIEW statement can be used only in interactive SQL.
- Only the owner of a view can remove that view.
- PROGRESS/SQL does not allow you to delete a view that is referenced in another view definition.

EDITING Phrase

Identifies the processing to take place following each keystroke during a PROMPT-FOR, SET, or UPDATE statement.

SYNTAX

```
[ label :] EDITING: statement...    END.
```

statement

One or more statements you want to process, usually for each keystroke entered. In most cases the first statement is READKEY.

EXAMPLES

r-edit.p

```
  DEFINE VARIABLE i AS INTEGER.
➡ UPDATE i EDITING:
    READKEY.
    APPLY LASTKEY.
  END.
```

This procedure lets you update the i variable, immediately processing each of your keystrokes. The READKEY statement reads each of the keys you press. The APPLY statement applies, or executes each keystroke. This is a very simple EDITING phrase and is the same as simply saying UPDATE i.

The r-edit2.p procedure uses an EDITING phrase with an UPDATE statement to control what happens based on each keystroke during the UPDATE. Here, the user can press any key while updating any field except sales-rep.

While in the sales-rep field, the user can press the space bar to scroll through the possible values for the sales-rep field. If the user presses the TAB, BACK-TAB, GO, RETURN, or END-ERROR key, the procedure executes that key. If the user presses any other key while in the sales-rep field, the terminal beeps.

r-edit2.p

```
PROMPT-FOR customer.cust-num.
FIND customer USING cust-num.
/* Update customer fields, monitoring each
   keystroke during the UPDATE */
UPDATE name address city st SKIP sales-rep HELP
       "Use the space bar to select a sales-rep"
        sales-region WITH 2 COLUMNS
  EDITING:
    /* Read a keystroke just pressed */
    READKEY.
    /* If the cursor is in any field except
       sales-rep, execute the last key pressed
       and go on to the next iteration of this
       EDITING phrase to check the next key */
    IF FRAME-FIELD < > "sales-rep" THEN DO:
      APPLY LASTKEY.
      IF GO-PENDING THEN LEAVE.
      ELSE NEXT.
    END.
    /* When in the sales-rep field, if the
       last key pressed was the space bar then
       cycle through the sales reps */
    IF LASTKEY = KEYCODE(" ") THEN DO:
      FIND NEXT salesrep NO-ERROR.
      IF NOT AVAILABLE salesrep
      THEN FIND FIRST salesrep.
      DISPLAY salesrep.sales-rep @ customer.sales-rep
              slsrgn @ sales-region.
      NEXT.
    END.
    /* If the user presses any one of a set
       of keys while in the sales-rep field,
       immediately execute that key */
    IF LOOKUP(KEYFUNCTION(LASTKEY),
            "TAB,BACK-TAB,GO,RETURN, END-ERROR") > 0
    THEN APPLY LASTKEY.
    ELSE BELL.
  END.
```

NOTES

- A READKEY statement does not have to be the first statement after the word EDITING. However, it should appear somewhere in the EDITING phrase because PROGRESS does not automatically read keystrokes when you use an EDITING phrase.

- The EDITING phrase applies to the PROMPT-FOR part of a SET or UPDATE statement. Therefore, to examine a value supplied by the user (within an EDITING phrase), you must use the INPUT function to refer to the field or variable containing the value.

- When you use the NEXT statement in an EDITING phrase, PROGRESS does the next iteration of that EDITING phrase and cancels any pending GO.

- When you use the LEAVE statement in an EDITING phrase, PROGRESS leaves the EDITING phrase and does the assignment part of the SET or UPDATE statement.

- Within an EDITING phrase, you cannot use the CLEAR ALL, DOWN, or UP statements on the frame being edited.

SEE ALSO PROMPT-FOR Statement, READKEY Statement, SET Statement, UPDATE Statement, Chapter 6 of the *Programming Handbook*

ENCODE Function

Encodes a source character string and returns the encoded character string result.

SYNTAX

```
ENCODE ( expression )
```

expression

A constant, field name, variable, or any combination of these which results in a character string value. If you use a constant, you must enclose it in quotation marks (" ").

EXAMPLE

r-encode.p

```
DEFINE VARIABLE password AS CHARACTER FORMAT "x(16)".
DEFINE VARIABLE id AS CHARACTER FORMAT "x(12)".
DEFINE VARIABLE n-coded-p-wrd AS CHARACTER FORMAT "x(16)".

SET id LABEL "Enter user id" password LABEL
    "Enter password" BLANK WITH CENTERED SIDE-LABELS.

n-coded-p-wrd = ENCODE(password).
DISPLAY n-coded-p-wrd LABEL "Encoded password".
```

This procedure uses the ENCODE function to scramble a password the user enters. The procedure then displays the encoded password.

NOTES

- The ENCODE function performs a one-way encoding operation which cannot be reversed. It is useful for storing scrambled copies of passwords in a database. It is impossible to determine the original password by examining the database. However, a procedure can prompt a user for a password, encode it, and compare the result with the stored, encoded password to determine if the user did supply the correct password.

- The output of the ENCODE function is 16 characters long. Remember to make sure the target field size is at least 16 characters long.

END Statement

Indicates the end of a block started with a DO, FOR EACH, or REPEAT statement or the end of an EDITING phrase.

SYNTAX

```
END
```

EXAMPLE

```
                                                      r-end.p
    FOR EACH customer:
      DISPLAY customer.cust-num name phone.
      FOR EACH order OF customer:
        DISPLAY order WITH 2 COLUMNS.
→     END.
→   END.
```

This procedure contains two blocks, each ending with the END statement.

NOTES

- If you use any END statements in a procedure, you must use one END statement for every block in the procedure.
- If you do not use any END statements in a procedure, PROGRESS assumes that all blocks end at the end of the procedure.

SEE ALSO DO Statement, EDITING Phrase, FOR EACH Statement, REPEAT Statement

ENTERED Function

Checks to see if a frame field has been modified during the last INSERT, PROMPT-FOR, SET, or UPDATE statement for that field and returns a TRUE or FALSE result.

SYNTAX

[FRAME *frame*] *field* ENTERED

field

The name of the field or variable you are checking.

EXAMPLE

r-enter.p

```
DEFINE VARIABLE new-max LIKE max-credit.
FOR EACH customer:
  DISPLAY cust-num name max-credit
          LABEL "Current max credit"
          WITH FRAME a 1 DOWN ROW 1.
  SET new-max LABEL "New max credit"
    WITH SIDE-LABELS NO-BOX ROW 10 FRAME b.
➡ IF new-max ENTERED THEN DO:
    IF new-max < > max-credit THEN DO:
    DISPLAY "Changing Max Credit of" name SKIP
            "from" max-credit "to"
            new-max WITH FRAME c ROW 15 NO-LABELS.
      max-credit = new-max.
      NEXT.
    END.
  END.
  DISPLAY "No Change in Max-Credit" WITH FRAME d ROW 15.
END.
```

This procedure goes through the customer file and prompts the user for a new max-credit value. The ENTERED function tests the value the user enters. If the user enters a new value, the procedure displays the old and new max-credit values. If the user enters the same or no value, it does not change the value.

NOTE

- If you type blanks in a field where data has never been displayed, the ENTERED function returns FALSE and a SET or ASSIGN statement does not update the underlying field or variable. Also, if PROGRESS has marked a field as entered, and the PROMPT-FOR statement prompts for the field again, and you do not enter any data, PROGRESS no longer considers the field entered.

SEE ALSO NOT ENTERED Function

ENTRY Function

Returns a character string entry from a list based on an integer position.

SYNTAX

```
ENTRY  ( element, list    )
```

element

An integer value corresponding to the position of a character string in a list of values. If the value of *element* does not correspond to an entry in the list, PROGRESS raises the ERROR condition. If the value of *element* is unknown (?), ENTRY returns an unknown value. If *element* is less than or equal to 0, or is larger than the number of elements in *list*, ENTRY returns an error.

list

A list of character strings. Separate entries with commas. If the value of *list* is unknown (?), ENTRY returns an unknown value.

EXAMPLE

r-entry.p

```
DEFINE VARIABLE datein AS DATE.
DEFINE VARIABLE daynum AS INTEGER.
DEFINE VARIABLE daynam AS CHARACTER INITIAL "Sunday,
   Monday, Tuesday, Wednesday, Thursday, Friday, Saturday".

SET datein LABEL "Enter a date (mm/dd/yy)".
daynum = WEEKDAY(datein).
DISPLAY ENTRY(daynum,daynam) FORMAT "x(9)"  ◄—
           LABEL "is a" WITH SIDE-LABELS.
```

This procedure returns the day of the week that corresponds to a date the user enters. The WEEKDAY function evaluates the date and returns, as an integer, the day of the week for that date. The ENTRY function uses that integer to indicate a position in a list of the days of the week.

SEE ALSO LOOKUP Function

EQ or = Operator

Returns a TRUE value if two expressions are equal.

SYNTAX

expression $\left\{ \begin{matrix} \text{EQ} \\ = \end{matrix} \right\}$ *expression*

expression

A constant, field name, variable name, or any combination of these. The expressions on either side of the EQ or = must be of the same data type, although one may be integer and the other decimal.

EXAMPLE

r-eq.p

```
PROMPT-FOR order.sales-rep WITH SIDE-LABELS CENTERED.

FOR EACH order
  WHERE sales-rep EQ INPUT sales-rep:
  DISPLAY order-num name odate pdate sdate  WITH CENTERED.
END.
```

This procedure prompts for the initials of a sales rep. The FOR EACH block reads the record for every sales rep that has initials equal to those supplied. The DISPLAY statement displays information from each of the retrieved records.

NOTES

- Most character comparisons are case-insensitive in PROGRESS. By default, all characters are converted to upper case prior to comparisons. However, it is possible to define fields and variables as case-sensitive (though it is not advised, unless case is significant.). If either *expression* is a field or variable defined to be case-sensitive, the comparison is case-sensitive. For example, "Smith" does not equal "smith."

- If one of the expressions has an unknown value and the other does not, the result is FALSE. If both are unknown, the result is TRUE. The exception to this is SQL. For SQL, if either or both expressions is unknown, then the result is unknown.

EXP Function

Returns a value resulting from raising a number to a power.

SYNTAX

```
EXP( base, exponent    )
```

base

A numeric expression (a constant, field name, variable name or any combination of these whose value is numeric).

exponent

A numeric expression.

EXAMPLE

r-exp.p

```
DEFINE VARIABLE principal AS DECIMAL
  FORMAT "->>>,>>9.99" LABEL "Amt Invested".
DEFINE VARIABLE rate AS INTEGER FORMAT "->9"
  LABEL "Interest %".
DEFINE VARIABLE num-yrs AS INTEGER FORMAT ">>9"
  LABEL "Number of Years".
DEFINE VARIABLE final-amt AS DECIMAL FORMAT
  "->>>,>>>,>>>,>>>,>>>,>>9.99" LABEL "Final Amount"
REPEAT:
  UPDATE principal rate num-yrs.
  final-amt =
➜   principal * EXP(1 + rate / 100,num-yrs).
  DISPLAY final-amt.
END.
```

This procedure calculates how much a principal amount invested at a given compounded annual interest rate will grow over a specified number of years.

NOTES

- After converting the base and exponent to floating-point format, the EXP function uses standard system library routines. On some machines, these routines do not handle very large numbers well and may cause your terminal to hang. Also, because the calculations are done in floating-point arithmetic, full decimal precision is not possible beyond 8–12 significant digits on most machines.

- The EXP function provides precision of about 15 decimal points.

EXPORT Statement

Converts data to a standard character format and displays it to the current output destination (except when the current output destination is the screen) or to a named output stream. Data exported to a file in standard format can be used as input to other PROGRESS procedures.

SYNTAX

```
EXPORT [ STREAM stream ] { expression ...
                         { record [EXCEPT field ...] }
```

STREAM *stream*

Specifies the name of a stream. If you do not name a stream, the unnamed stream is used. See the DEFINE STREAM reference page and Chapter 9 of the *Programming Handbook* for more information about streams.

expression

One or more expressions (an expression is a constant, field name, variable name or any combination of these) that you want to convert into standard character format for display to an output destination.

record

The name of the record buffer whose fields you want to convert into standard character format to display to an output destination.

To use EXPORT with a record in a file defined for multiple databases, you must qualify the record's filename with the database name. See the description of the Record Phrase for more information.

EXCEPT *field*

All fields except those fields listed in the EXCEPT phrase will be exported.

EXAMPLE

r-exprt.p

```
OUTPUT TO customer.d.
FOR EACH customer:
  EXPORT customer.
END.
```

This procedure converts the data in the customer file into standard character format and sends that data to the customer.d file.

NOTES

- EXPORT must follow an OUTPUT TO statement. Other procedures can later use the exported data as input by reading the file with INSERT, PROMPT-FOR, SET, UPDATE or IMPORT statements, naming one field or variable to correspond to each data element.

- The data is in a *standard format* to be read back into PROGRESS. All character fields are enclosed in quotes ("") and quotes contained in the data you are exporting are replaced by two quotes (""). A single space separates one field from the next. An unknown value is displayed as an unquoted question mark.

- Each EXPORT statement creates one line of data. That is, fields are not wrapped onto several lines. For example:

```
OUTPUT TO custdump.
FOR EACH customer:
    EXPORT cust-num name max-credit.
END.
```

- This procedure creates an ASCII file, custdump, with one line for each customer. A typical line has the following format:

```
1 "Second Skin Scuba" 1500
```

- There are no trailing blanks, leading zeros, or formatting characters (e.g. dollar signs) in the data.

- PROGRESS exports logical fields as the value "yes" or "no".

- A format phrase with an EXPORT statement is ignored.

- If you use a single qualified identifier with the EXPORT statement, the compiler first interprets the reference as *dbname.filename*. If the compiler cannot resolve the reference as *dbname.filename*, it tries to resolve it as *filename.fieldname*.
- When exporting fields, you must use filenames that are different from field names to avoid ambiguous references. See the description of the Record Phrase for more information.
- When exporting recid fields, you must explicitly state the recid field name in the EXPORT statement.
- EXPORT is sensitive to the Date format (-d), Century (-yy), and European numeric (-E) startup options. When loading data with the IMPORT statement, use the same settings that you used with the EXPORT statement.

SEE ALSO DEFINE STREAM Statement, DISPLAY Statement, INPUT FROM Statement, IMPORT Statement, PUT Statement, STRING Function

FETCH Statement (SQL)

Retrieves the next row from the retrieval set accessed by the OPEN statement.

SYNTAX

FETCH *cursor-name* INTO *variable-list*

Cursor-name
The name given to the cursor when defined in the CURSOR statement.

INTO *variable-list*
Lists the procedure variables to receive the column values.

EXAMPLE

```
FETCH c03 INTO pname, pnum.
```

NOTE

- The FETCH statement can be used in both interactive SQL and embedded SQL.

FILL Function

Generates a character string made up of a character string repeated a specified number of times.

SYNTAX

```
FILL ( expression, repeats       )
```

expression

A constant, field name, variable name, or any combination of these whose value is character.

repeats

A constant, field name, variable name, or any combination of these whose value is an integer. The FILL function uses this value to repeat the *expression* you specify. If the value of *repeats* is less than or equal to 0, FILL produces a null string.

EXAMPLE

r-fill.p

```
DEFINE VARIABLE percentg AS INTEGER FORMAT ">>9".
DEFINE VARIABLE fillchar AS CHARACTER  FORMAT "x".

fillchar = IF OPSYS = "msdos"
           THEN "~333"
           ELSE "*".

FOR EACH customer:
  ACCUMULATE ytd-sls(TOTAL).
END.

DISPLAY "Percentage of Company Sales" WITH CENTERED NO-BOX.

FOR EACH customer:
  percentg = ytd-sls / (ACCUM TOTAL ytd-sls) * 100.
  FORM SKIP(1) name percentg LABEL "%" bar AS CHAR LABEL
    " 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17"
    FORMAT "x(50)" WITH NO-BOX NO-UNDERLINE.
  COLOR DISPLAY BRIGHT-RED bar.
➤ DISPLAY name percentg FILL(fillchar,percentg * 3) @ bar.
END.
```

This procedure produces a bar chart depicting the sales to each customer as a percentage of total company sales. The first FOR EACH block accumulates the value of ytd-sls for each customer, producing a total year-to-date sales value for the company. The next FOR EACH block goes through the customer file again, figuring each customer's year-to-date sales as a percentage of the total.

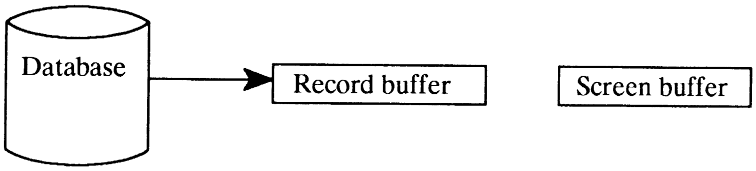
The FORM statement describes the frame layout, including the name, the percentage of total sales, and then a bar across the top of the frame. The variable named bar is defined "on-the-fly". That is, it has no corresponding DEFINE VARIABLE statement at the top of the procedure. It is defined right in the FORM statement, and has its own label and format. The DISPLAY statement following the FORM statement displays the bar variable. If the procedure is running on UNIX or VMS, or on a monochrome PC monitor, the COLOR BRIGHT-RED is ignored. However, if you are running the procedure on a PC with a color monitor, the bar is displayed in BRIGHT-RED (a predefined color on the PC).

The final DISPLAY statement displays the bars themselves. The "fillchar = ..." statement describes which fill character to use. On a PC, the bar character is a solid box (defined by the character represented by octal 333); on UNIX and VMS, the bar character is an asterisk. The FILL function generates a string made up of fill characters that is the percentage of total sales multiplied by 3 (each percentage point uses three fill characters).

FIND Statement

Uses an index to locate a single record in a file and moves the record into a record buffer.

DATA MOVEMENT



SYNTAX

```
     [ FIRST ]
FIND [ LAST  ]  record-phrase  [ NO-WAIT ]  [ NO-ERROR ]
     [ NEXT  ]
     [ PREV  ]
```

FIRST

Finds the first record in the file that meets the characteristics you may have specified with *record-phrase*. If the buffer named in the *record-phrase* was preselected in a DO or REPEAT statement, FIND locates the first record in that preselected subset of records.

LAST

Finds the last record in the file that meets the characteristics you may have specified with *record-phrase*. If the buffer named in the *record-phrase* was preselected in a DO or REPEAT statement, FIND locates the last record in that preselected subset of records.

NEXT

Finds the next record in the file that meets the characteristics you may have specified with *record-phrase*. If no record has yet been found, FIND NEXT behaves like FIND FIRST. If the buffer named in the *record-phrase* was preselected in a DO or REPEAT statement, the FIND locates the next record in that preselected subset of records.

PREV

Finds the previous record in the file. If no record has yet been found, FIND PREV behaves like FIND LAST. If the buffer named in the *record-phrase* was preselected in a DO or REPEAT statement, the FIND finds the previous record in that preselected subset of records.

record-phrase

Identifies the record you want to retrieve.

Here is the syntax for *record-phrase*:

```
record [ constant ] [ WHERE expression               ] ...
                    [ USING field [ AND field ]...   ]
                    [ OF file                        ]
                    [ USE-INDEX index                ]

                    [ SHARE-LOCK     ]
                    [ EXCLUSIVE-LOCK ]
                    [ NO-LOCK        ]
```

For more information on *record-phrase*, see the Record Phrase reference page.

To find a record in a file defined for multiple databases, you must qualify the record's filename with the database name. See the description of the Record Phrase for more information.

For more information on *record-phrase* and qualifying database names, see the Record Phrase reference page.

NO-WAIT

Causes FIND to return immediately and raise an error condition if the record is locked by another user (unless you use the NO-ERROR option on the same FIND statement).

If you are using the FIND statement to find a record in a work file, PROGRESS disregards the NO-WAIT option.

NO-ERROR

Tells PROGRESS not to display error messages for any errors it may encounter. Possible errors include not finding a record that satisfies the *record-phrase* or finding more than one record that satisfies the *record-phrase*. If you use the NO-ERROR option, you can also use the AVAILABLE function to test whether FIND actually found a record.

EXAMPLES

```
                                                   r-find.p

  REPEAT:
    PROMPT-FOR item.item-num.
➤   FIND item USING item-num.
    DISPLAY item-num idesc.
    REPEAT:
      FIND NEXT order-line OF item.
➤     FIND order OF order-line.
➤     FIND customer WHERE customer.cust-num = order.cust-num.
      DISPLAY customer.name order.order-num
              order-line.qty (TOTAL).
    END.
  END.
```

This procedure produces a report that shows, for an item, all the customers that bought an item, and in what quantity. The procedure finds an item record, the order-lines that use that item, the order associated with each order-line and, finally, the customer associated with each order.

```
                                                   r-find2.p

  DEFINE VARIABLE start-name LIKE customer.name.
  REPEAT:
    SET start-name.
➤   FIND FIRST customer WHERE name >= start-name.
    REPEAT:
      DISPLAY name.
➤     FIND NEXT customer USE-INDEX name.
    END.
  END.
```

The FIND FIRST statement finds the first record whose name field value is greater than or equal to the string supplied by the user. The FIND NEXT statement finds the next record in the file, using the name index.

NOTES

- Before doing a FIND and storing the record in the record buffer, PROGRESS clears the record buffer. Before clearing the record buffer, PROGRESS validates the record in the buffer. If the record fails validation, you see an error message.

- If you define a buffer other than the default buffer in which to store records that are read, the *record-spec* or *record* you name with the FIND statement can be the name of that defined buffer.

- A FIND statement that does not supply FIRST, LAST, NEXT, or PREV must be able to locate at most one record based solely on the conditions in the index it is using and ignoring any non-index field selection criteria specified in the record-spec. When it locates the record, it applies any non-index field selection criteria to that record, and supplies the record if it meets the criteria.

- If a FIND NEXT or FIND PREV does not find any more records, PROGRESS takes the endkey action. By default, this action is UNDO, LEAVE for a FOR EACH, REPEAT, or procedure block.

- See the DEFINE BUFFER statement for a description of using FIND on a PRESELECTed set of records.

- If there is exactly one customer record, the following statement finds that record. Otherwise, the following statement always returns “not found” rather than “ambiguous”:

```
FIND customer WHERE name BEGINS "".
```

- Also, the following statement succeeds if there is only one “Smith” in the database:

```
FIND customer WHERE last-name = "Smith"
  AND first-name BEGINS "".
```

- When you use the FIND statement, PROGRESS selects an index to use based on the WHERE condition or the USE-INDEX option. If you supply conditions other than exact equality on the leading index field components, those conditions become “selection criteria” and are tested once the record is retrieved.

  The FIND statement must locate either one or no records based solely on the conditions in the index lookup, ignoring the selection criteria. If one record is found, then the selection criteria (if any) are applied to that record and the FIND is successful if it passes.

For example, suppose you have a file named "test" and it contains fields a, b, and c. The file is indexed jointly on a,b, and c, in that order. Imagine you have the following records:

a	b	c
1	1	2
1	1	3
2	3	4

Given this information, look at the following FIND statements:

```
FIND test WHERE a = 1 AND b = 1 AND c = 2.
/* This procedure succeeds */
```

```
FIND test WHERE a = 1 AND b = 1 AND c = 1.
/* This procedure fails because no such record
   exists */
```

```
FIND test WHERE a = 1 AND c = 2.
/* This procedure fails because there are two
   records with a = 1 in the index and the c = 2
   criterion is not applied once more than one
   record matches the leading index component(s)
   criteria.  The result is no record found
   (AVAILABLE test is FALSE) and AMBIGUOUS test
   is TRUE. */
```

```
FIND test WHERE a = 2.
/* This procedure succeeds */
```

```
FIND test WHERE a = 1 AND b = 1 AND c >= 3.
/* This procedure fails because the criteria is
   ambiguous based on the leading index component
   equality matches of a = 1 and b = 1. */
```

See the Notes section of the FOR statement for more information on how PROGRESS chooses what index to use.

- Your position in an index is established when you find a record and is only modified by subsequent finds, not by CREATEs or by changing index field values.

- If you are using the FIND statement to find a record in a workfile, you must use the FIRST, LAST, NEXT, or PREV option with the FIND statement.

- In a REPEAT block, if you use the FIND NEXT statement to find a record and then do an UNDO, RETRY of a block, the FIND NEXT statement reads the next record in the file rather than the one found in the block iteration where the error occurred. (PROGRESS does an UNDO, RETRY if there is an error if you explicitly use the UNDO, RETRY statement, or if you press END-ERROR on the second or later screen interaction in a block.)

- For example:

```
REPEAT:
  FIND NEXT order.
  DISPLAY order.
  SET order-num.
  SET odate pdate.
END.
```

Here, if you press END-ERROR during the second SET statement, PROGRESS displays the next record in the file.

If you are using a FOR EACH block to read records, and do an UNDO, RETRY during the block, you will see the same record again rather than the next record.

If you want to use a REPEAT block and want to see the same record in the event of an error, use the following statements:

```
REPEAT:
  IF NOT RETRY THEN FIND NEXT order.
  DISPLAY order.
  SET order-num.
  SET odate pdate.
END.
```

- When you use FIND NEXT or FIND PREV to find a record after updating another record, be careful that you do not lose your updates in the event that the record you are trying to find is unavailable. For example:

```
FIND FIRST customer.
REPEAT:
  UPDATE customer.
  FIND NEXT customer.
END.
```

In this example, if the FIND NEXT statement fails to find the customer record, any changes made during the UPDATE statement are undone. To avoid this situation, use the following technique:

```
FIND FIRST customer.
REPEAT:
  UPDATE customer.
  FIND NEXT customer NO-ERROR.
  IF NOT AVAILABLE customer THEN LEAVE.
END.
```

- After you use the FIND LAST statement to find the last record in a file, PROGRESS positions the index cursor on that record. Any references within the same record scope to the next record will fail. For example:

```
FIND LAST customer.
RELEASE customer.
DISPLAY AVAILABLE customer.
REPEAT:
  FIND NEXT customer.
  DISPLAY name.
END.
```

In this example, the RELEASE statement releases the last customer record from the customer record buffer and the following DISPLAY statement displays FALSE because the customer record is no longer available. However, the index cursor is still positioned on that last record. Therefore, the FIND NEXT statement fails.

- If you use FIND *recid*... on a PRESELECTed (see DO, REPEAT statements) list of records, the index cursor is not reset. That is, FIND NEXT does not find the record that follows record *recid* in the preselected list.

- Here is some important information to remember when using a FIND NEXT or FIND PREV statement in a subprocedure to access a record from a shared buffer:

  When you run a PROGRESS procedure, PROGRESS creates an cursor indicator for each file accessed in the procedure and each NEW buffer defined in the procedure. A cursor indicator serves as an anchor for index cursors associated with a file or buffer. An index cursor is attached to cursor indicator when you enter a block of code to which a record buffer is scoped. If two different indexes are used for the same record buffer within a single block of code, there are two index cursors attached to the same cursor indicator. When the program control leaves the block to which a record buffer is scoped, all index cursors attached to the cursor indicator go away.

  When PROGRESS encounters a subprocedure in a procedure, it first checks through the existing index cursors before creating any other index cursors needed by the statements in the subprocedure.

  If the use-index of the FIND NEXT/PREV statement in a subprocedure accesses an index cursor for a shared buffer that existed prior to the beginning of the subprocedure, the FIND NEXT/PREV statement returns next/prev record for the shared buffer based upon the last record found in that buffer and the use-index of the FIND statement.

  If the use-index of the FIND NEXT/PREV statement in a subprocedure accesses an index cursor created for a shared buffer at the beginning of the subprocedure, the FIND NEXT/PREV statement returns first/last record for the shared buffer based upon the use-index of the FIND statement.

SEE ALSO AMBIGUOUS Function, AVAILABLE Function, DEFINE BUFFER Statement, FOR EACH Statement, LOCKED Function, NEW Function

FIRST Function

Returns a TRUE value if the current iteration of a DO, FOR EACH, or REPEAT...BREAK block is the first iteration of that block.

SYNTAX

```
FIRST( break-group )
```

break-group

The name of a field or expression you name in the block header with the BREAK BY option.

EXAMPLE

r-first.p

```
DEFINE VARIABLE order-value AS DECIMAL  LABEL "Order-value".
FOR EACH order:
  DISPLAY order-num.
  FOR EACH order-line OF order
    BREAK BY qty * price:
➤   IF FIRST(qty * price)
    THEN order-value = 0.
    order-value = order-value + qty * price.
    DISPLAY item-num line-num qty * price
            LABEL "Extended price".
  END.
  DISPLAY order-value.
END.
```

The r-first.p procedure displays, for each order record, the order number, order-lines on the order, the extended price of each order-line, and a total order value for the order.

Because the inner FOR EACH block iterates until the order-lines of an order are all read, the procedure must set the order-value variable to 0 each time there is a new order being used in that block. The FIRST function uses the (qty * price) expression as the *break-group* to keep track of whether or not the current iteration is the first iteration of the FOR EACH block.

SEE ALSO FIRST-OF Function, LAST Function, LAST-OF Function

FIRST-OF Function

Returns a TRUE value if the current iteration of a DO, FOR EACH, or REPEAT...BREAK block is the first iteration for a new break group.

SYNTAX

```
FIRST-OF(break-group )
```

break-group
The name of a field or expression you name in the block header with the BREAK BY option.

EXAMPLE

```
                                              r-firstf.p

  FOR EACH item BREAK BY prod-line:
➔   IF FIRST-OF(prod-line) THEN CLEAR ALL.
    DISPLAY prod-line item-num idesc.
  END.
```

This procedure generates a report that lists all the item records grouped by product line. When the product line changes, the procedure clears the current list of items and displays items belonging to the new product line. The FIRST-OF function uses the value of the prod-line field to determine when that value is different from the value during the last iteration.

NOTE

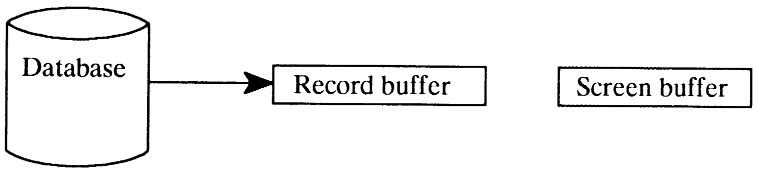
- If you are doing calculations in a block, you use the BREAK option to tell PROGRESS to do certain calculations when the value of certain expressions changes. PROGRESS uses default formatting to display the results of these calculations. If you want to control the formatting, you can use the FIRST-OF function to determine the start of a break group and then do the formatting.

SEE ALSO FIRST Function, LAST Function, LAST-OF Function

FOR Statement

Starts an iterating block that reads a record from each of one or more files at the start of each block iteration.

DATA MOVEMENT



BLOCK PROPERTIES

Iteration, Record Reading, Record Scoping, Frame Scoping, Transactions by default.

SYNTAX

```
[ label : ]
   FOR [ EACH  ] record-phrase  [ , [ EACH  ] record-phrase ] ...
       [ FIRST ]                [   [ FIRST ]               ]
       [ LAST  ]                [   [ LAST  ]               ]

        [ [ BREAK ] { BY expression  [ DESCENDING ] } ... ]
        [ variable = expression1   TO expression2   [ BY k ] ]
        [ WHILE expression  ]
        [ TRANSACTION ]
        [ ON ENDKEY-phrase ]
        [ ON ERROR-phrase ]
        [ frame-phrase  ]
```

EACH

Starts an iterating block, finding a single record on each iteration. If you do not use the EACH keyword, the Record phrase you use must identify exactly one record in the file.

FIRST

Uses the criteria in the *record-phrase* to find the first record in the file that meets that criteria. PROGRESS finds the first record before doing any sorting. For example:

```
FOR FIRST customer BY max-credit:
  DISPLAY customer.
END.
```

This procedure displays customer 1 (cust-num is the primary index of the customer file), not the customer with the lowest max-credit. A procedure that displays the customer with the lowest max-credit might look like this:

```
FOR EACH customer BY max-credit:
  DISPLAY customer.
  LEAVE.
END.
```

See the Notes section for more information on using this option.

LAST
Uses the criteria in the *record-phrase* to find the last record in the file that meets that criteria. PROGRESS finds the last record before doing any sorting. For example:

```
FOR LAST customer BY max-credit:
  DISPLAY customer.
END.
```

This procedure displays the customer with the highest customer number (cust-num is the primary index of the customer file), not the customer with the lowest max-credit. A procedure that displays the customer with the lowest max-credit might look like this:

```
FOR EACH customer BY max-credit:
  DISPLAY customer.
  LEAVE.
END.
```

See the Notes section for more information on using this option.

record-phrase
Identifies the set of records you want to retrieve.

To use FOR EACH/FIRST/LAST to access a record in a file defined for multiple databases, you must qualify the record's filename with the database name. See the description of the Record Phrase for more information.

Here is the syntax for *record-phrase*:

```
record  [ constant ]  ┌ WHERE expression                  ┐
                      │ USING field [ AND field ]...      │ ...
                      │ OF file                           │
                      └ USE-INDEX index                   ┘

                      ┌ SHARE-LOCK     ┐
                      │ EXCLUSIVE-LOCK │
                      └ NO-LOCK        ┘
```

For more information on *record-phrase*, see the Record Phrase reference page.

BREAK

Over a series of block iterations, you may want to do some work based on whether the value of a certain field changes. This field defines a "break group." For example, you might be accumulating some value, such as a total. You would use the BREAK option to define st as the break group. For example:

```
FOR EACH customer BREAK BY st:
  DISPLAY st name max-credit (TOTAL BY st).
END.
```

Here, PROGRESS accumulates the total max-credit for all the customers in the customer file. Each time the value of the st field changes, PROGRESS displays a subtotal of the max-credit values for customers in that state.

You can use the BREAK option anywhere in the block header but you must also use the BY option to name a sort field.

You can use the BREAK option in conjunction with the ACCUMULATE statement and ACCUM function. For more information, see the reference pages for these functions.

BY *expression* [DESCENDING]

Sorts the selected records by the value of *expression*. If you do not use the BY option, records are retrieved in order of the index used to satisfy the *record-phrase* criteria, or the primary index if no criteria are given. The DESCENDING option sorts the records in descending order (not in default ascending order).

You can use multiple BY options to do multi-level sorting. For example:

```
FOR EACH customer BY max-credit BY name
```

Here, the customers are sorted in order by max-credit. Within each max-credit value, customers are sorted alphabetically by name.

NOTE: There is a performance benefit if an index on *expression* exists: BREAK BY does not need to perform the sort that is otherwise required to evaluate FIRST, LAST, FIRST-OF, and LAST-OF expressions.

variable = *expression1* TO *expression2* [BY *k*]

The name of a field or variable whose value you are incrementing in a loop. *expression1* is the starting value for *variable* on the first iteration of the loop. *k* is the amount to add to *variable* after each iteration and must be a constant. When *variable* exceeds *expression2* (or is less than *expression2* if *k* is negative) the loop ends. Because *expression1* is compared to *expression2* at the start of the first iteration of the block, the block may be executed 0 times. *expression2* is reevaluated on each iteration of the block.

WHILE *expression*

Indicates the condition under which you want the FOR EACH block to continue processing the statements within it. Using the WHILE *expression* option causes the block to iterate as long as the condition specified by the expression is true or the end of the index being scanned is reached, whichever comes first. The expression is any combination of constants, field names, and variable names that yields a logical value.

TRANSACTION

Identifies the FOR EACH block as a system transaction block. PROGRESS starts a system transaction for each iteration of a transaction block if there is not already an active system transaction. See Chapters 5 and 8 of the *Programming Handbook* for more information about transactions.

ON ENDKEY-*phrase*

Describes the processing that takes place when the ENDKEY condition occurs during a block. Here is the syntax for the ON ENDKEY-*phrase:*

```
                                 ┌                      ┐
                                 │ , LEAVE  [ label2 ]  │
ON ENDKEY UNDO [ label1 ]        │ , NEXT   [ label2 ]  │
                                 │ , RETRY  [ label2 ]  │
                                 │ , RETURN             │
                                 └                      ┘
```

For more information about ON ENDKEY-*phrase*, see the ON ENDKEY Phrase reference page.

ON ERROR-*phrase*

Describes the processing that takes place when there is an error during a block.

Here is the syntax for the ON ERROR-*phrase:*

```
                              ┌ , LEAVE   [ label2 ] ┐
ON ERROR  UNDO[ label1 ]      │ , NEXT    [ label2 ] │
                              │ , RETRY   [ label2 ] │
                              └ , RETURN             ┘
```

For more information about ON ERROR-*phrase*, see the ON ERROR Phrase reference page.

frame-phrase

Specifies the overall layout and processing properties of a frame. Here is the syntax of *frame-phrase*:

```
      ┌                                                              ┐
      │ ACCUM                                                        │
      │ ATTR-SPACE                                                   │
      │ CENTERED                                                     │
      │        { [ DISPLAY ] color-phrase }                          │
      │ COLOR  { PROMPT color-phrase      }  ...                     │
      │ COLUMN  expression                                           │
      │ n COLUMNS                                                    │
      │ DOWN                                                         │
      │ expression    DOWN                                           │
      │ FRAME frame                                                  │
WITH  │ NO-ATTR-SPACE                                                │  ...
      │ NO-BOX                                                       │
      │ NO-HIDE                                                      │
      │ NO-LABELS                                                    │
      │ NO-UNDERLINE                                                 │
      │ NO-VALIDATE                                                  │
      │ OVERLAY                                                      │
      │ PAGE-BOTTOM                                                  │
      │ PAGE-TOP                                                     │
      │ RETAIN n                                                     │
      │ ROW  expression                                              │
      │ SCROLL  n                                                    │
      │ SIDE-LABELS                                                  │
      │ TITLE [ COLOR   color-phrase     ] expression                │
      │ TOP-ONLY                                                     │
      │ WIDTH  n                                                     │
      └                                                              ┘
```

For more information on *frame-phrase*, see the Frame Phrase reference page.

EXAMPLES

```
r-fore.p

FOR EACH customer WHERE cust-num < 12 BY st:
  DISPLAY cust-num name address city st.
END.
```

This procedure reads those customer records whose cust-num is less than 12, sorting the records in order by state before displaying them.

```
r-fore2.p

FOR EACH customer, EACH order OF customer,
  EACH order-line OF order, item OF order-line
  BY pdate BY customer.cust-num BY line-num:
  DISPLAY pdate customer.cust-num  order.order-num
          line-num item.item-num idesc.
END.
```

This procedure gets information from four related files (customer, order, order-line, and item) and displays some information from each. Before displaying the information, the FOR EACH statement sorts it in order by the pdate field, then, within that field, in order by cust-num. Within the cust-num field, the data is sorted by the line-num field.

```
r-fore3.p

FOR EACH customer, LAST order OF customer:
  DISPLAY customer.cust-num customer.name order.order-num
          order.odate order.misc-info.
  PAUSE 1 NO-MESSAGE.
  misc-info = "Last order".
  DISPLAY misc-info.
END.
```

This procedure uses the LAST option to display information about the last order of each customer.

NOTES

- At compile time, PROGRESS determines which index to use for retrieving records from a file based on the conditions in the Record phrase. PROGRESS selects only one index for each file referenced in a particular FOR EACH statement.

- An index component is involved in an **equality match** if it is used in the Record phrase conditions in the form:

```
field = expression
```

where the expression is independent of any fields in the file for which an index is being selected. A condition involving OF and USING are equivalent to this form. A field is involved in a **range match** if it is used in a condition of this form:

$$\text{field} \begin{bmatrix} < \\ <= \\ > \\ >= \\ \text{BEGINS} \end{bmatrix} \text{expression}$$

An equality or range match is considered **active** if the equality or range condition stands on its own or is related to other conditions solely through the AND operator (i.e. not through OR or NOT).

A field is involved in a **sort match** if it is used in a BY option of the form:

```
BY field [DESCENDING]
```

- The following list describes the rules the PROGRESS database manager uses to choose an index for a PROGRESS database.

1. If you specify the RECID of the record then PROGRESS accesses the record directly without using an index.

2. If, in the *record-phrase*, you use the USE-INDEX option, PROGRESS uses the index you name in that option.

3. For each index in the file, PROGRESS looks at each index component in turn and counts the number of active equality, range, and sort matches. PROGRESS ignores the counts for any components of an index that occur after a component that has no active equality match. PROGRESS compares the results of this count and selects the best index. PROGRESS uses the following order to determine the better of any two indexes:

   A. If one index is unique and all of its components are involved in active equality matches and the other index is not unique, or if not all of its components are involved in active equality matches, PROGRESS chooses the former of the two.

   B. Select the index with more active equality matches.

   C. Select the index with more active range matches.

   D. Select the index with more active sort matches.

   E. Select the index that is the primary index.

   F. Select the first index alphabetically by index name.

- PROGRESS may need to scan all the records in the index to find those meeting the conditions, or PROGRESS may need to examine only a subset of the records. This latter case is called **bracketing** the index and results in more efficient access. Having selected an index as described above, PROGRESS examines each component as follows to see if the index can be bracketed:

1. If the component has an active equality match then it can be bracketed and the next component is also examined for possible bracketing.

2. If the component has an active range match then it can be bracketed, but the remaining components are not examined for possible bracketing.

3. If the component has neither an active equality match nor an active range match, the remaining components are not examined for bracketing.

- Any conditions you specify in the *record-phrase* that are not involved in bracketing the selected index are applied to the fields in the record itself to determine if the record meets the overall *record-phrase* criteria. For example, assume that the f file has fields a,b, and c involved in two indexes:

  Primary, unique index (`I1`) on a,b, and c

  Secondary non-unique index (`I2`) on c

  Table 13 indicates, for various *record-phrases*, which index PROGRESS selects and what part of the index is bracketed.

Table 13: PROGRESS Index Selection Examples

Record-Phrase	Index Selected	Bracketing On
f WHERE a = 3 AND b = 2 AND c = 3	I1	a + b + c
f WHERE a = 3	I1	a
f WHERE c = 1	I2	c
f WHERE a = 3 AND b > 7 AND c = 3	I1	a + b
f WHERE a = 3 AND c = 4	I1	a
f WHERE b = 5	I1	None of the fields (1)
f WHERE a = 1 OR b > 5	I1	None of the fields (1)
f WHERE (a > = a1 AND a < = a2) OR (a1 = 0)	I1	None of the fields (2)
f WHERE a > = (IF a1 NE 0 THEN a1 ELSE -99999999) AND a < = (IF a1 NE O THEN a2 ELSE +99999999)	I1	a (2)

(1) In this case, PROGRESS must look at all of the records to determine which meet the specified criteria.

(2) The two record phrases in these examples are almost identical in effect, but the one using the OR operator to connect conditions is much less efficient in its use of the selected index.

- The FIRST and LAST keywords are especially useful when you are sorting records in a file about which you want to display information. Often several related records exist in a related file, but you only want to display the first or last related record from that file in the sort. You can use FIRST or LAST in these cases. Two examples follow.

```
FOR EACH customer, FIRST order OF customer:
  DISPLAY order.cust-num odate.
END.
```

Suppose you were interested in displaying the date each customer first placed an order. This procedure displays the customer number and date of the first order.

```
DISPLAY "Show the last order-line of each order," SKIP
       "sorted by the item's cost and the" SKIP
       "promised date of the order." WITH CENTERED.

FOR EACH order, LAST order-line OF order,
       item OF order-line by item.cost by order.pdate:
DISPLAY order.order-num line-num item.item-num cost
       pdate WITH TITLE "For FIRST/LAST" CENTERED.
END.
```

This procedure displays the last order line of every order, sorted by the cost of the item and by the promised date of the order.

SEE ALSO FIND Statement, Frame Phrase, Record Phrase, Chapters 5, 7, and 8 of the *Programming Handbook*

FORM Statement

Defines the layout and certain processing attributes of a frame. Use the FORM statement if you want to describe a frame in a single statement rather than let PROGRESS construct the frame based on individual data handling statements in a block.

All PROGRESS statements that display or update data can use a frame described by a FORM statement. These statements are DISPLAY, INSERT, PROMPT-FOR, SET, UPDATE, and COLOR.

SYNTAX

```
FORM
 ┌                              ┐
 │ field [ format-phrase ]      │
 │          ┌ AT n ┐            │
 │ constant └ TO n ┘            │ ...
 │ SPACE[( n )]                 │
 │ SKIP[( n )]                  │
 └                              ┘
 ┌        ┌            ┌ AT n            ┐     ┐ ┐
 │        │ expression │ TO n            │ ... │ │
 │ HEADER<             └ FORMAT string   ┘      > ... │ [ frame-phrase ]
 │        │ SPACE[( n )]                       │ │
 └        └ SKIP[( n )]                        ┘ ┘
```

```
FORM record   [ EXCEPT field ...]   [ frame-phrase ]
```

record

The name of the field or variable for which you are describing display characteristics.

To use the FORM statement to display a record in a file defined for multiple databases, you must qualify the record's filename with the database name. See the description of the Record Phrase for more information.

format-phrase

Specifies one or more frame attributes for a field, variable, or expression.

Here is the syntax for *format-phrase*:

```
[ AT n                                              ]
[ AS datatype                                       ]
[ ATTR-SPACE                                        ]
[ AUTO-RETURN                                       ]
[ BLANK                                             ]
[ COLON n                                           ]
[ COLUMN-LABEL  label  [ ! label ]...               ] ...
[ DEBLANK                                           ]
[ FORMAT string                                     ]
[ HELP string                                       ]
[ LABEL string                                      ]
[ LIKE field                                        ]
[ NO-ATTR-SPACE                                     ]
[ NO-LABEL                                          ]
[ TO n                                              ]
[ VALIDATE ( condition, msg-expression)             ]
```

For more information on Format phrase, see the Format Phrase reference page.

constant

A constant value.

AT *n*

The number (*n*) of the column in which you want to start the display. The AT option does not left justify the data; it indicates the placement of the data area.

TO *n*

The number (*n*) of the column in which you want the display to end. The TO option does not right justify the data; it simply indicates the placement of the data area.

SPACE (*n*)

Identifies the number (*n*) of blank spaces to be inserted after the expression is displayed. *n* can be zero (0). If the number of spaces you specify is more than the spaces left on the current line of the frame, a new line is started and extra spaces discarded. If you do not use this option and you do not use *n*, one space is inserted between items in the frame.

SKIP (*n*)

Identifies the number (*n*) of blank lines to be inserted after the expression is displayed. *n* can be zero (0). If you do not use this option, PROGRESS does not skip a line between expressions unless the expressions do not fit on one line. If you use the SKIP option but do not specify *n*, or if *n* is 0, PROGRESS starts a new line unless it is already at the beginning of a new line.

HEADER

Indicates that the specified items are to be placed in a header section at the top of the frame. This option tells PROGRESS to reevaluate all expressions in the FORM statement at the beginning of each new page. For example, page numbers are reevaluated at the start of every new page, thus the value changes.

When you use the FORM statement with the HEADER option, PROGRESS disregards Dictionary field labels for fields you name in the FORM statement. Use character strings to specify labels on fields you name in a FORM HEADER statement.

expression

A constant, field name, variable name, or any combination of these whose value you want to display in the frame header. PROGRESS evaluates *expression* each time the frame used by the FORM statement comes into view or is printed at the top or bottom of a page (if the frame is a PAGE-TOP or PAGE-BOTTOM frame). For example:

r-eval.p

```
DEFINE VARIABLE i AS INTEGER FORMAT ">9".
FORM HEADER "This is the header - i is" i
            WITH FRAME a ROW i COLUMN i i DOWN.
DO i = 1 TO 8 WITH FRAME a:
  DISPLAY i.
  PAUSE.
END.
```

The FORM statement defines a HEADER frame that consists of the text "This is the header - i is" and the value of the variable i. In addition, it also specifies a screen location where the header should be displayed. The FORM statement does not bring the header frame into view.

On the first iteration of the DO block, the DISPLAY statement brings the frame into view. On the second iteration of the DO block, the frame is already in view (it was not hidden during the first iteration), so the header of the frame is not reevaluated. That means that the new value of i is not reflected in the header portion of the frame and you do not see the new value of i in the header, nor do you see the position of the frame on the screen change.

In contrast, look at this modified version of the procedure:

```
r-eval2.p

DEFINE VARIABLE i AS INTEGER FORMAT ">9".
FORM HEADER "This is the header - i is"
            WITH FRAME a ROW i COLUMN i i DOWN.

DO i = 1 TO 8 WITH FRAME a:
  DISPLAY i.
  HIDE FRAME a.
END.
```

On the first iteration of the DO block, the DISPLAY statement brings the frame into view. The HIDE statement removes the frame from the screen. Therefore, on the second iteration of the DO block, the DISPLAY statement must bring the frame back into view. PROGRESS reevaluates the header of the frame each time the frame is brought into view. Therefore, the header of the frame reflects the change to i, and the position of the frame on the screen changes as well.

FORMAT *string*

The format in which you want to display the *expression*. If you do not use the FORMAT option, PROGRESS uses the defaults shown in Tables 14.

Table 14: Default Display Formats

Type of Expression	Default Format
Field	Format from Dictionary
Variable	Format from variable definition
Constant character	Length of character string
Other	Default format for the data type of the expression. Table 15 shows these default formats.

Table 15: Default Data Type Display Formats

Data Type of Expression	Default Format
Character	x(8)
Date	99/99/99
Decimal	->>,>>9.99
Integer	->,>>>,>>9
Logical	yes/no

For more information on data formats, see Chapter 4 of the *Programming Handbook.*

frame-phrase

Specifies the overall layout and processing properties of a frame. Here is the syntax for *frame-phrase*:

```
       [ ACCUM                                                ]
       [ ATTR-SPACE                                           ]
       [ CENTERED                                             ]
       [         { [ DISPLAY ] color-phrase }                 ]
       [ COLOR   { PROMPT color-phrase      } ...             ]
       [ COLUMN expression                                    ]
       [ n COLUMNS                                            ]
       [ DOWN                                                 ]
       [ expression   DOWN                                    ]
       [ FRAME frame                                          ]
WITH   [ NO-ATTR-SPACE                                        ]
       [ NO-BOX                                               ] ...
       [ NO-HIDE                                              ]
       [ NO-LABELS                                            ]
       [ NO-UNDERLINE                                         ]
       [ NO-VALIDATE                                          ]
       [ OVERLAY                                              ]
       [ PAGE-BOTTOM                                          ]
       [ PAGE-TOP                                             ]
       [ RETAIN n                                             ]
       [ ROW expression                                       ]
       [ SCROLL n                                             ]
       [ SIDE-LABELS                                          ]
       [ TITLE [ COLOR color-phrase ] expression              ]
       [ TOP-ONLY                                             ]
       [ WIDTH n                                              ]
```

For more information on *frame-phrase*, see the Frame Phrase reference page.

record

The name of the record you want to display. Naming a record is a shorthand way of listing each field individually.

EXCEPT *field*

All fields except those fields listed in the EXCEPT phrase shown in the form.

EXAMPLE

```
r-form.p

REPEAT FOR customer:

  FORM name    COLON 10  phone     COLON 50
       address COLON 10  sales-rep COLON 50 SKIP
       city    COLON 10 NO-LABEL st NO-LABEL
       zip NO-LABEL WITH SIDE-LABELS 1 DOWN
       CENTERED.

  PROMPT-FOR cust-num WITH FRAME cnum
     SIDE-LABELS CENTERED.
  FIND customer USING cust-num.
  UPDATE name address city st zip phone sales-rep.
END.
```

This procedure lets the user update information about a specific customer. The FORM statement describes a very specific layout for the UPDATE statement to use.

When you use the FORM statement to control the order in which fields appear on the screen, remember that this order is independent of the order in which the fields are processed during data entry.

Here, the FORM statement displays the customer name first and the phone number second. But the UPDATE statement specifies the phone number after the name, address, city, st, and zip. The fields are displayed as described in the FORM statement, but the user can access the fields only in the order described by the UPDATE statement.

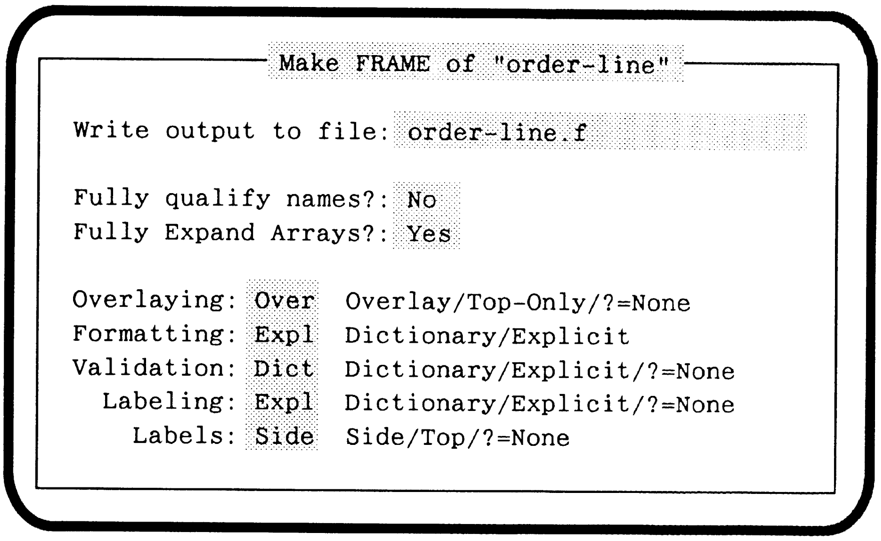
NOTES

- When you use any of the statements that access the screen, you can either name a frame or use the default frame for the block in which the statements appear. For more information on frame scoping, see Chapter 7 of the *Programming Handbook*.
- When PROGRESS compiles a procedure, it makes a top to bottom pass of the procedure to design all the frames for that procedure, including those referenced in FORM statements. PROGRESS adds field and format attributes as it goes through the procedure.

- If you use a single qualified identifier with the FORM statement, the compiler first interprets the reference as *dbname.filename*. If the compiler cannot resolve the reference as *dbname.filename*, it tries to resolve it as *filename.fieldname*.

  When naming fields in a FORM statement, you must use filenames that are different from field names to avoid ambiguous references. See the description of the Record Phrase for more information.

- You can use the Data Dictionary to automatically generate FORM statements based on the fields in a file. Enter the Data Dictionary. Press **U** (Utilities), **G** (Generate Include Files), and **F** (FORM statement). Select a file with the arrow keys or by typing its first few letters, and then press RETURN. The "Make FRAME" prompt (shown below) appears; use it to prepare your FORM statement.



```
──────────── Make FRAME of "order-line" ────────────

 Write output to file: order-line.f

 Fully qualify names?: No
 Fully Expand Arrays?: Yes

 Overlaying: Over  Overlay/Top-Only/?=None
 Formatting: Expl  Dictionary/Explicit
 Validation: Dict  Dictionary/Explicit/?=None
   Labeling: Expl  Dictionary/Explicit/?=None
     Labels: Side  Side/Top/?=None
```

For example, below is an include file that includes an FORM statement for the fields in the file `order-line` from the PROGRESS demo database. Note that the field names are fully qualified--the only variation from the default choices in the Make FRAME prompt. Now you can use the PROGRESS editor, or any text editor, to add or delete fields, to tailor the formatting, or to make any other changes to the FORM statement. This is especially useful for files with numerous record fields.

```
/*order-ln.f*/
/* 10/10/89 FORM LIKE file order-line */
FORM
 order-line.Order-num FORMAT ">>>>9" LABEL "Order num"
 order-line.Line-num FORMAT ">>9" LABEL "Line num"
 order-line.Item-num FORMAT "99999" LABEL "Item num"
 order-line.Price FORMAT "->,>>>,>>9.99" LABEL "Price"
 order-line.Qty FORMAT "->>>>9" LABEL "Qty"
 order-line.Qty-ship FORMAT "->,>>>,>>9" LABEL "Qty ship"
 order-line.Disc FORMAT ">>9" LABEL "Disc %"
 WITH FRAME order-line OVERLAY SIDE-LABELS.
```

- Below is an alternative FORM statement for the `order-line` file. It is different from the FORM statement above only because different choices are made at the "Make FRAME" prompt.

```
/* order-ln.f */
/* 10/10/89 FORM LIKE file order-line */

FORM
  Order-num VALIDATE(CAN-FIND(order OF order-line),
                                        "Order must exist")
  Line-num
  Item-num VALIDATE(CAN-FIND(item OF order-line),
                                    "Item must be on file")
  Price
  Qty
  Qty-ship
  Disc
  WITH FRAME order-line TOP-ONLY.
```

```
 Overlaying: Top   Overlay/Top-Only/?=None          (No
 Formatting: Dict  Dictionary/Explicit             SIDE-LABELS)
 Validation: Expl  Dictionary/Explicit/?=None
   Labeling: Dict  Dictionary/Explicit/?=None
     Labels: Top   Side/Top/?=None
```

No FORMAT *display-format* or LABEL *label* supplied here; display format and labels will appear in the frame as defined in the Data Dictionary.

Note the choices made in the "Make FRAME" prompt and their effects on the resulting FORM statement. Because we selected explicit validation, the generated FORM statement explicitly includes any validation clauses defined in the Data Dictionary. Likewise, DISPLAY formats and LABELs defined in the Dictionary appear in the first FORM statement, but not in the second one. Choosing to include FORMAT, LABEL, or VALIDATE clauses explicitly makes it easier for you to go in and change them.

SEE ALSO Format Phrase, Frame Phrase

Format Phrase

Specifies one or more frame attributes for a field, variable, or expression.

SYNTAX

```
[ AT n                                        ] ...
[ AS datatype                                 ]
[ ATTR-SPACE                                  ]
[ AUTO-RETURN                                 ]
[ BLANK                                       ]
[ COLONn                                      ]
[ COLUMN-LABELlabel [ ! label   ]...          ]
[ DEBLANK                                     ]
[ FORMATstring                                ]
[ HELPstring                                  ]
[ LABELstring                                 ]
[ LIKE field                                  ]
[ NO-ATTR-SPACE                               ]
[ NO-LABEL                                    ]
[ TO n                                        ]
[ VALIDATE (condition, msg-expression   )     ]
```

AT *n*

The number (*n*) of the column in which you want the display to start. The AT option does not left justify the data; it simply indicates the placement of the data area.

AS *datatype*

Creates a frame field and variable with the data type you specify. This is useful for defining display positions in a frame for use with DISPLAY @.

ATTR-SPACE

There are two ways a terminal can handle screen formatting. It either:

- Reserves a character position, on both sides of every field, for special screen field attributes, such as underlining or highlighting. These are called "spacetaking" terminals.
- Does not reserve a character position for special field attributes. These are called "nonspacetaking" terminals.

Specifying ATTR-SPACE reserves spaces in the frames used by the procedure for field attributes such as underlining and highlighting. See Chapter 7 of the *Programming Handbook* for more information.

AUTO-RETURN

When you enter the last character in the field, PROGRESS automatically moves out of the field as if you had pressed RETURN. If this happens on the last field of a data entry statement, then PROGRESS behaves as if you pressed GO.

For the purposes of AUTO-RETURN, entering leading zeros in a numeric field does not count as "filling" the field. For example, suppose you define a numeric field as follows:

```
DEFINE VARIABLE x AS INTEGER FORMAT "99".
SET x AUTO-RETURN.
```

If you enter a 09 into the field, PROGRESS does not AUTO-RETURN. To get the AUTO-RETURN behavior in this situation, define the field as CHARACTER with a format of "99".

BLANK

Displays blanks for the field you are displaying or entering. This is useful for entering passwords.

COLON *n*

The number (*n*) of the column in which you want the colon of the label to appear. Use this option with SIDE-LABEL frames where the labels are placed to the left of the data and are separated from the data with a colon.

COLUMN-LABEL *label* [! *label*] ...

Names the label you want to display above the field. If you want the label to use more than one line ("stacked" labels), use a separate *label* for each line and separate each *label* with a exclamation point (!). For example:

```
r-colbl2.p

FOR EACH customer:
➤  DISPLAY name COLUMN-LABEL "Customer!Name"
➤     sales-rep COLUMN-LABEL "Name of!Sales!Representative".
END.
```

PROGRESS does not display column labels if you use the SIDE-LABELS or NO-LABELS option with the frame phrase.

You must enclose the label string in quotation marks. If you want to use the exclamation point (!) as one of the characters in a column label, you must use two exclamation points ("!!").

DEBLANK
Removes leading blanks (for use on input character fields only).

FORMAT *string*
The format in which you want to display the *expression*. You must enclose the string in quotation marks (""). If you do not use the FORMAT option, PROGRESS uses the defaults shown in these tables.

Table 16: Default Display Formats

Type of Expression	Default Format
Field	Format from Dictionary
Variable	Format from variable definition
Constant character	Length of character string
Other	Default format for the data type of the expression. Table 17 shows these default formats.

Table 17: Default Data Type Display Formats

Data Type of Expression	Default Format
Character	x(8)
Date	99/99/99
Decimal	->>,>>9.99
Integer	->,>>>,>>9
Logical	yes/no

You can use the FORMAT option with the UPDATE and SET statements to store a character string that is longer than the field length you define in the Data Dictionary or in a DEFINE VARIABLE statement. This is possible because PROGRESS stores data in variable-length fields. For example:

```
DEFINE VARIABLE mychar AS CHARACTER FORMAT "x(3)".
UPDATE mychar FORMAT "x(8)".
```

You can also use the ASSIGN statement to store data in a field or variable that is longer than the predefined format of that field or variable:

```
        mychar = "abcdefgh".
```

However, the Data Dictionary load program only loads character data that is **no longer** than the format you defined in the Dictionary.

For more information on data formats, see Chapter 4 of the *Programming Handbook.*

HELP *string*

A character string you want to display whenever the user enters the frame field for the field or variable. When the user leaves the frame field, PROGRESS removes the help string from the message area. You must enclose the character string in quotation marks ("").

If the input source is not the terminal, PROGRESS disregards any HELP options.

LABEL *string*

A character string you want to use as a label for a field, variable, or expression. You must enclose the string in quotation marks (""). Table 18 shows the order PROGRESS uses to determine the label for a field, variable, or expression.

Table 18: Determining Labels

	LABEL *string*	**Dictionary Label**	**Field Name**	**LIKE** *field*	**Variable Name**
Field	1	2	3	N/A	N/A
Variable	1	N/A	N/A	2	3
Expression	1	N/A	N/A	N/A	N/A

LIKE *field*

Creates a frame field and variable with the same definition as *field.*

The LIKE option in a DEFINE VARIABLE statement, DEFINE WORKFILE statement, or format phrase requires that a particular database be connected. Since you can startup a PROGRESS application session without connecting to a database, you should use the LIKE option with caution.

NO-ATTR-SPACE

Does not reserve spaces in the frames used by the procedure for field attributes such as underlining and highlighting. See the Chapter 7 of the *Programming Handbook* for more information.

NO-LABEL
Prevents PROGRESS from displaying a label for a field, variable, or expression.

TO *n*
The number (*n*) of the column in which you want to end the display. The TO option does not right justify the data; it indicates the placement of the data area.

VALIDATE(*condition,msg-expression*)
You use the VALIDATE option to specify a value against which you want to validate data entered into a screen field or variable.

condition is a boolean expression (a constant, field name, variable name, or any combination of these) whose value is either TRUE or FALSE. When you use the VALIDATE option to validate a specific field, any reference to that field in *condition* is assumed to be an input field. For example, the pdate field is assumed as input in the following statement:

```
SET odate pdate VALIDATE(pdate > odate,
  "Promise date must be later than order date").
```

The previous statement is equivalent to:

```
SET odate pdate VALIDATE(INPUT pdate > odate,
  "Promise date must be later than order date").
```

The validation is based on the value of odate prior to the SET statement. If you want to validate the value of pdate against the input value of odate, use this statement:

```
SET odate pdate VALIDATE(pdate > INPUT odate,
  "Promise date must be later than order date").
```

If you attempt to validate a field whose reference is ambiguous, PROGRESS tries to resolve the ambiguity by referencing the file that contains the record being updated. In the following example, the address2 field is ambiguous because it exists in both the order file and the customer file. PROGRESS resolves the ambiguity by validating the address2 field in the order file, since the order file is being updated.

```
FIND FIRST customer.
FIND FIRST order.
UPDATE order.cust-num order.address order.address2
  VALIDATE(address2 EQ "" AND INPUT address2 NE "",
    "Place the one-line address on the first address line).
```

If the reference is to an array field and has no subscript, PROGRESS assumes you want to use the subscript of the field being prompted for.

msg-expression is the message you want to display if the value of *condition* is FALSE. You must enclose *msg-expression* in quotation marks (“”).

PROGRESS processes validation criteria whenever the user attempts to leave the frame field. If the frame field value is not valid, PROGRESS displays *msg-expression* in the message area, causes the terminal to “beep,” and does not advance out of the frame field.

If a user tabs into a frame field, makes no changes, and leaves the field, PROGRESS does not process the validation criteria specified with the VALIDATE option until the you press GO (F1). If you press ENDKEY or END-ERROR (F4), or an error occurs, PROGRESS does not test the validation criteria specified with the VALIDATE option.

If the input source for the procedure is a file, PROGRESS validates each input field (except those with a value of “-”). If the result of the validation is FALSE, *msg-expression* is displayed and PROGRESS treats the validation as an error.

To suppress the Dictionary validation criteria for a field, use the following VALIDATE option:

```
VALIDATE(TRUE,"")
```

When you use the VALIDATE option in a procedure to specify validation criteria for a field, that validation criteria applies to all other references to that field in the same frame. For example:

```
FOR EACH order:
  UPDATE odate.
  UPDATE odate VALIDATE(odate LE TODAY,
    "Can't be later than today").
END.
```

In this example, PROGRESS applies the validation criteria on the second UPDATE statement also to the first UPDATE statement because both UPDATE statements use the same frame. Scope references to the same field to different frames if you do not want a VALIDATE option to affect all references to that field.

EXAMPLE

r-frmat.p

```
DEFINE VARIABLE password AS CHARACTER.

UPDATE password FORMAT "x(6)" BLANK
  VALIDATE(password = "secret","Sorry, wrong password")
  HELP "Maybe the password is 'secret'!"
  WITH FRAME passw CENTERED SIDE-LABELS.
HIDE FRAME passw.

REPEAT:
  PROMPT-FOR customer.cust-num COLON 20.
  FIND customer USING cust-num.
  UPDATE
    name     LABEL "Customer Name" COLON 20
             VALIDATE(name ne "","Please enter a name")
    address  HELP"Please enter two lines of address"
             COLON 20 LABEL "Address"
    address2 NO-LABEL COLON 22
    city     COLON 20
    st       COLON 20
    zip      COLON 20 SKIP(3)
➤   phone    AT 5 FORMAT "(999) 999-9999"
    contact  TO 60
    WITH CENTERED SIDE-LABELS.
END.
```

This procedure lets the user update customer records after entering the password of "secret." The Format phrase on the phone field describes the display format of that field.

NOTES

- The ATTR-SPACE/NO-ATTR-SPACE designation in a Frame Phrase takes precedence over an ATTR-SPACE/NO-ATTR-SPACE designation in a Format Phrase. The ATTR-SPACE/NO-ATTR-SPACE designation in a Format Phrase takes precedence over an ATTR-SPACE/NO-ATTR-SPACE designation in a COMPILE statement.

SEE ALSO FORM Statement, Frame Phrase

Frame Phrase

Specifies the overall layout and processing properties of a frame. When used on block header statements (DO, FOR EACH, and REPEAT), the Frame phrase also specifies the default frame for data handling statements (DISPLAY, SET, etc.) within the block. Frame phrases can also be used on individual statements to indicate the specific frame to which the statement applies.

SYNTAX

```
       ┌ ACCUM                                                  ┐
       │ ATTR-SPACE                                             │
       │ CENTERED                                               │
       │       ┌ [ DISPLAY ] color-phrase ┐                     │
       │ COLOR ┤                          ├ ...                 │
       │       └ PROMPT color-phrase      ┘                     │
       │ COLUMNexpression                                       │
       │ n COLUMNS                                              │
       │ DOWN                                                   │
       │ expression  DOWN                                       │
       │ FRAME frame                                            │
       │ NO-ATTR-SPACE                                          │
       │ NO-BOX                                                 │
WITH   │ NO-HIDE                                                │ ...
       │ NO-LABELS                                              │
       │ NO-UNDERLINE                                           │
       │ NO-VALIDATE                                            │
       │ OVERLAY                                                │
       │ PAGE-BOTTOM                                            │
       │ PAGE-TOP                                               │
       │ RETAIN n                                               │
       │ ROW expression                                         │
       │ SCROLL n                                               │
       │ SIDE-LABELS                                            │
       │ TITLE [ COLORcolor-phrase ] expression                 │
       │ TOP-ONLY                                               │
       └ WIDTHn                                                 ┘
```

ACCUM

The ACCUM option allows you to use aggregate functions (such as MAX, MIN, TOTAL, and SUBTOTAL) to accumulate values within shared frames. With the ACCUM option, aggregate values can be shared among procedures through shared frames. You must include the ACCUM option in the FORM statement of each procedure that uses the shared frame, as shown in the following examples:

```
/* testa.p */

DEFINE NEW SHARED FRAME x.
FORM field1 field2 WITH FRAME x ACCUM.
RUN testb.p.
```

```
/* testb.p */

DEFINE SHARED FRAME x.
FORM field1 field2 WITH FRAME x ACCUM.
FOR EACH file1:
 DISPLAY field1 field2 (TOTAL) WITH FRAME x.
END.
```

ATTR-SPACE

There are two ways a terminal can handle screen formatting. It either:

- Reserves a character position on both sides of every field for special screen field attributes, such as underlining or highlighting. These terminals are called "spacetaking" terminals.
- Does not reserve a character position for special field attributes. These are called "nonspacetaking" terminals.

Specifying ATTR-SPACE reserves spaces in the frames used by the procedure for field attributes such as underlining and highlighting. See Chapter 7 of the *Programming Handbook* for more information.

CENTERED

Centers the frame on the terminal screen. If you use the CENTERED option and are sending output to a device other than the terminal, PROGRESS centers the frame for the terminal. This may result in a non-centered frame on the alternate output device.

You may also use the COLUMN option to indicate the position of the frame.

COLOR *color-phrase* [INPUT *color-phrase*]

Specifies a video attribute or color to use for the background of the frame, and optionally, a video attribute or color to use for data fields enabled for input.

The *color-phrase* option specifies a video attribute or color. The default color for the background of a frame is NORMAL. Here is the syntax for *color-phrase*:

```
{ NORMAL
  INPUT
  MESSAGES
  protermcap-attribute
  dos-hex-attribute
  [ BLINK- ][BRIGHT-][ fgnd-color ] [ / bgnd-color ]
  [ BLINK- ][RVV- ][ UNDERLINE- ][BRIGHT- ][fgnd-color ]
  VALUE(expression)
}
```

For more information about *color-phrase*, see the Color Phrase reference page.

If you are using DOS or OS/2, be sure to indicate both a background and a foreground color in the *color-phrase* part of the BACKGROUND option. The background color is the background color of the frame and the foreground color is the color of all labels and headers in the frame.

Here is a procedure that uses the COLOR option on the frame phrase.

r-bkcol.p

```
FOR EACH customer:
  DISPLAY cust-num name max-credit WITH 10 DOWN FRAME cust
          CENTERED TITLE "Customer Credit Information".
  IF max-credit >= 1500
  THEN DO:
    UPDATE max-credit
          LABEL "Max credit too high. Please change"
          WITH SIDE-LABELS FRAME x COLOR MESSAGES
          PROMPT MESSAGES ROW 17 CENTERED.
    DISPLAY max-credit WITH FRAME cust.
  END.
END.
```

In the r-bkcol.p procedure PROGRESS displays customers and their maximum credit. If the credit is above $1,500, PROGRESS displays frame x with the maximum credit message and a prompt for the user to change the maximum credit. PROGRESS displays the entire frame in the color MESSAGES.

In the UPDATE statement of the r-bkcol.p procedure, the frame phrase for frame x includes the option COLOR MESSAGES. This option tells PROGRESS to display frame x in the color MESSAGES. The PROMPT MESSAGES option tells PROGRESS to display the prompt in the color MESSAGES also.

COLUMN *expression*

The *expression* is a constant, field name, variable name or any combination of these whose value is the number of the column in which you want to start a frame. PROGRESS ignores this option if you also use the CENTERED option for the same frame.

PROGRESS evaluates *expression* each time the frame comes into view or is printed at the top or bottom of a page (if the frame is a PAGE-TOP or PAGE-BOTTOM frame). For more details, see the *expression* option of the FORM statement.

n COLUMNS

Formats data fields into a specific number (*n*) of columns. Truncates labels to 16, 14, and 12 characters when the number of columns is 1, 2, or 3 respectively. PROGRESS reserves a fixed number of positions in each column for labels. For *n* = 1, 18 positions are allowed for a label; for *n* = 2, 16 positions are allowed; and for *n* = 3, 14 positions are allowed. Label positions include room for a colon and a space after the label. Labels are right justified if they are short and truncated if they are too long.

When you use this option, it implies SIDE-LABELS and overrides any AT, COLON, TO, or SPACE options you may have used in the same Frame phrase.

DOWN

Displays multiple records in a single frame. For more information on DOWN frames, see Chapter 7 of the *Programming Handbook.*

expression DOWN

The *expression* is a constant, field name, variable name or any combination of these whose value is the number of repetitions you want in a frame.

PROGRESS evaluates *expression* each time the frame comes into view or is printed at the top or bottom of a page (if the frame is a PAGE-TOP or PAGE-BOTTOM frame). For more details, see the *expression* option of the FORM statement.

FRAME *frame*

You can define new frames by giving them unique names. Whenever the same frame name is referred to in more than one Frame phrase, the characteristics on each Frame phrase naming that frame are combined. Also, any frame characteristics used in data handling statements that name the same frame are combined into the same frame description.

NO-ATTR-SPACE

Does not reserve spaces in the frames used by the procedure for field attributes such as underlining and highlighting. See Chapter 7 of the *Programming Handbook* for more information.

NO-BOX

Does not display a box around the frame. If you do not use this option, PROGRESS displays a box around the data you are displaying.

If you are sending data to a device other than a terminal and you do not use this option, PROGRESS omits the sides and bottom line of the box and replaces the top line with blanks.

NO-HIDE

Tells PROGRESS to suppress the automatic hiding of the frame (when the block to which the frame is scoped iterates). The frame is hidden only if space is needed to display other frames.

NO-HIDE suppresses hiding for a frame only when the block to which that frame is scoped iterates. For example:

```
FOR EACH customer:
  DISPLAY cust-num name.
  FOR EACH  order OF customer:
    DISPLAY order.order-num.
    DISPLAY "hello" WITH FRAME b COLUMN 60 NO-HIDE.
  END.
END.
```

In this example, PROGRESS does not hide frame b on iterations of the inner block. However, it does hide frame b when the outer block iterates. If you want the frame to stay in view during iterations of the outer block, scope the frame to that block.

NO-LABELS

Does not display labels. This option overrides any COLUMN-LABEL option you include in another phrase or statement.

NO-UNDERLINE

Does not underline labels appearing above fields.

NO-VALIDATE

Tells PROGRESS to disregard all validation conditions specified in the Dictionary for fields entered in this frame.

OVERLAY

Indicates that the frame can overlay any other frame that does not use the TOP-ONLY option. If you do not use this option, the frame you are using cannot overlay other frames. If PROGRESS needs to bring an OVERLAY frame into view and doing so will partially obscure a TOP-ONLY frame, it first hides the TOP-ONLY frame.

Here is a procedure that uses the OVERLAY option on the Frame Phrase.

r-ovrlay.p

```
FOR EACH customer:
  DISPLAY customer WITH 2 COLUMNS
          TITLE "Customer Information".
  FOR EACH order OF customer:
    DISPLAY order WITH 2 COLUMNS OVERLAY
            TITLE "Customer's Orders" ROW 7 COLUMN 10.
  END.
END.
```

This procedure displays customer information in one frame. The procedure then displays order information for the customer in a second frame that overlays the first.

PAGE-BOTTOM

Displays the frame at the bottom of the page each time the output ends a page.

PAGE-TOP

Displays the frame each time the output begins on a new page.

Table 19 shows how the PAGE-TOP and PAGE-BOTTOM options work depending on the kind of DISPLAY or VIEW you are doing.

Table 19: Using PAGE-TOP And PAGE-BOTTOM Frames

	OUTPUT to aPAGED file	**OUTPUT to screen or an UNPAGED file**
DISPLAY a PAGE-TOP frame	Thc framc is writtcn to thc file and put on a list of frames to be written at the top of each page.	The frame is written to the file or displayed on the screen.
VIEW a PAGE-TOP frame	The frame is put on a list of frames to be written at the top of each page.	The frame is written to the file or displayed on the screen.
DISPLAY a PAGE-BOTTOM frame	The frame is written to the file and put on a list of frames to be written at the bottom of each page.	The frame is written to the file or displayed on the screen.
VIEW a PAGE-BOTTOM frame	The frame is put on a list of frames to be written at the bottom of each page.	The frame is written to the file or displayed on the screen.
HIDE either type	The frame is removed from the appropriate list.	The frame is removed from the screen.

RETAIN *n*

Specifies the number of frame iterations to retain when the frame scrolls on the screen. *n* must be a constant. For example, RETAIN 2 causes the last 2 iterations in a down frame to be displayed at the top of the frame. If you are using UP to scroll up a screen, those two lines are painted at the bottom of the screen. Do not use the SCROLL option in a frame phrase in which you also use the RETAIN option.

ROW *expression*

The *expression* is a constant, field name, variable name or any combination of these whose value is the row in which you want to start a frame. If you are displaying a frame on a device other than a terminal, this option has no effect.

PROGRESS evaluates *expression* each time the frame comes into view or is printed at the top or bottom of a page (if the frame is a PAGE-TOP or PAGE-BOTTOM frame). For more details, see the *expression* option of the FORM statement.

SCROLL *n*

Displays a scrolling rather than a paging frame. *n* specifies the number of frame iterations to scroll when the frame scrolls on the screen. *n* must be a constant. For example, if a procedure uses a DISPLAY or DOWN statement when a scrolling frame is full, the data in the frame scrolls up *n* iterations (rather than clearing and repainting the frame as it would without the SCROLL option).

Here is a procedure that uses the SCROLL option to scroll the display one line at a time.

```
r-fphrsc.p

FOR EACH customer WHERE cust-num <= 30:
  DISPLAY cust-num name max-credit WITH SCROLL 1.
  IF max-credit >= 1000
  THEN COLOR DISPLAY MESSAGES max-credit.
END.
```

Do not use the RETAIN option in a frame phrase in which you also use the SCROLL option.

SIDE-LABELS

Displays field labels to the left of the data and separated from the data by a colon and a space. If you do not use the SIDE-LABELS option, PROGRESS displays labels above their corresponding fields and separates the labels from the field values with underlining.

STREAM *stream*

Allows you to specify the name of a stream for SQL statements.

TITLE [COLOR *color-phrase*] *expression*

Displays a title as part of the top line of the box around a display frame. *expression* is a constant, field name, variable name, or any combination of these whose result is a character value. *expression* is the value you want to display as a title. If *expression* is a constant character string, it must be surrounded by quotes (""). PROGRESS automatically centers *expression* in the top line of the frame box.

The *color-phrase* option specifies a video attribute or color. The default color for the TITLE is MESSAGES. Here is the syntax for *color-phrase*:

```
{ NORMAL                                                              }
{ INPUT                                                               }
{ MESSAGES                                                            }
{ protermcap-attribute                                                }
{ dos-hex-attribute                                                   }
{ [ BLINK- ][BRIGHT-][fgnd-color ] [ / bgnd-color  ]                  }
{ [ BLINK- ][RVV- ][ UNDERLINE- ][BRIGHT- ] [ fgnd-color]             }
{  VALUE ( expression )                                               }
```

For more information about *color-phrase*, see the Color Phrase reference page.

TOP-ONLY

Indicates that no other frame can overlay this frame. If you do not use this option, other frames that use the OVERLAY option can overlay this frame. If PROGRESS needs to bring an OVERLAY frame into view and by doing so will partially obscure a TOP-ONLY frame, it first hides the TOP-ONLY frame. For more information, see the OVERLAY option.

WIDTH *n*

Specifies the number (*n*) of columns in a frame. If you do not use the WIDTH option, the width of the frame is based on the fields you are displaying and the width of the terminal you are using.

EXAMPLES

r-frame.p

```
    FOR EACH customer:
      FORM HEADER
            "No-Box,No-Underline,No-Labels,5 Down"
             SKIP "Centered" SKIP(2)
➛            WITH NO-BOX NO-UNDERLINE NO-LABELS
                  CENTERED 5 DOWN.
      DISPLAY cust-num name phone.
    END.
```

The r-frame.p procedure displays the cust-num, name, and phone number for each customer record. The Frame phrase (starting with the word WITH) describes the frame being used to display that information.

```
r-frame2.p

OUTPUT TO phone.lst PAGED PAGE-SIZE 20.

FOR EACH customer:
  FORM HEADER "Customer List" AT 1 PAGE-NUMBER
            TO 60 WITH FRAME hdr PAGE-TOP
            CENTERED NO-BOX.
  VIEW FRAME hdr.
  FORM "Customer List Continued On Next Page"
       WITH FRAME footr PAGE-BOTTOM CENTERED.
  VIEW FRAME footr.

  DISPLAY cust-num name phone WITH CENTERED.

END.
OUTPUT CLOSE.
```

The r-frame2.p procedure produces a customer report, using "Customer List" as the header for each page of the report and using "Customer List Continued On Next Page" as the footer for each page of the report. The OUTPUT TO statement directs all output to the file phone.lst. After running the r-frame2.p procedure, you can look at the contents of the phone.lst file by pressing GET (F5) and then typing the name of the file, phone.lst.

NOTES

- PAGE-TOP and PAGE-BOTTOM frames are activated based on DISPLAY or VIEW statements as described above. They are deactivated when the block to which the frames are scoped iterates or ends.

- You can do input and output to a frame only when that frame is in full view. Therefore, when you do input or output to a frame that is hidden or partially overlayed, PROGRESS brings that frame into view before doing any input or output to the frame.

- The ATTR-SPACE/NO-ATTR-SPACE designation in a Frame Phrase takes precedence over an ATTR-SPACE/NO-ATTR-SPACE designation in a Format Phrase. The ATTR-SPACE/NO-ATTR-SPACE designation in a Format Phrase takes precedence over an ATTR-SPACE/NO-ATTR-SPACE designation in a COMPILE statement.

- An empty WITH clause is legal. If the WITH keyword appears by itself, or in the clause following an earlier WITH, it is ignored. This feature is useful when designing template programs to be called with arguments. For example, a template program with a line like DISPLAY {1} WITH {2} executes correctly even if called with only one argument.

SEE ALSO FRAME-COL Function, FRAME-DOWN Function, FRAME-LINE Function, FRAME-ROW Function, FORM Statement, Format Phrase, Chapter 7 of the *Programming Handbook.*

FRAME-COL Function

Returns an integer value that is the column position of the upper left corner of a frame.

SYNTAX

```
FRAME-COL [ (frame ) ]
```

frame

The name of the frame whose column position you are trying to determine. If you do not supply a frame name, the FRAME-COL function uses the default frame for the block it is in. If the FRAME-COL function is in a DO block, the function uses the default frame scoped to the block containing the DO block.

EXAMPLE

r-frcol.p

```
FOR EACH customer:
  DISPLAY customer WITH FRAME cust-frame 2 COLUMNS
          TITLE "CUSTOMER INFORMATION".
  FOR EACH order OF customer:
    DISPLAY order-num odate sdate pdate shp-via misc-info
            cust-po shipped WITH 2 COLUMNS 1 DOWN OVERLAY
            TITLE "CUSTOMER'S ORDERS"
            ROW FRAME-ROW(cust-frame) + 8
            COLUMN FRAME-COL(cust-frame) + 1.
  END.
END.
```

This procedure displays customer information in one frame, then displays order information in an overlay frame. FRAME-ROW places the overlay frame on the eighth row of the first frame. FRAME-COL places the overlay frame on the first column of the first frame.

NOTE

- The FRAME-COL function returns a value of 0 if the frame you specify is not in view when the function is evaluated.

SEE ALSO Frame Phrase, FRAME-DOWN Function, FRAME-LINE Function, FRAME-ROW function

FRAME-DB Function

Returns the logical database name for the field in which the cursor was last positioned for input.

SYNTAX

```
FRAME-DB
```

The function requires no arguments.

If the cursor was last positioned in a field that is not a database field, this function displays no value for the field.

EXAMPLE

r-frdb.p

```
FOR EACH customer:
  UPDATE cust-num name address address2 city
      st zip WITH 1 DOWN 1 COLUMN CENTERED
      EDITING:
        DISPLAY
        "You are editing field:" FRAME-FIELD SKIP
        "Of file:" FRAME-FILE SKIP
➤       "In database:" FRAME-DB
        WITH FRAME a ROW 15 NO-LABELS CENTERED.
        READKEY.
        APPLY LASTKEY.
      END. /* Editing */
  END.
```

For each field being updated, this procedure displays the field name, the file the field belongs to, and the database in which the file exists. The EDITING phrase is part of the UPDATE statement; it displays information about the field as you update the record, and then reads each of the keystrokes entered (READKEY) and applies those keystrokes (APPLY LASTKEY).

NOTES

- If the cursor is not in an enabled input field when the last input statement is executed, FRAME-DB returns an empty string.
- Use this syntax to find the name of a schema-holder for a foreign database: `SDBNAME(FRAME-DB)`.

SEE ALSO FRAME-FIELD Function, FRAME-FILE Function, LDBNAME Function, FRAME-INDEX Function, PROGRAM-NAME Function.

FRAME-DOWN Function

Returns an integer value that represents the number of iterations in a frame.

SYNTAX

```
FRAME-DOWN [ (frame ) ]
```

frame

The name of the frame whose number down you are trying to determine. If you do not supply a frame name, the FRAME-DOWN function uses the default frame for the block it is in. If the FRAME-DOWN function is in a DO block, the function uses the default frame scoped to the block containing the DO block.

EXAMPLE

r-frdown.p

```
DEFINE VARIABLE ans AS LOGICAL.

REPEAT:
  FIND NEXT customer.
  DISPLAY cust-num name.
  IF FRAME-LINE = FRAME-DOWN
  THEN DO:
    MESSAGE "Do you want to see the next page ?"
    UPDATE ans.
    IF NOT ans
    THEN LEAVE.
  END.
END.
```

This procedure displays customers in a frame. When the frame is full, the procedure asks, "Do you want to see the next page ?" The procedure knows the frame is full when the value of FRAME-LINE (current logical line number) equals the value of FRAME-DOWN (number of iterations in the frame).

NOTE

- The FRAME-DOWN function returns a value of 0 if used with a single frame or if the frame is not in view when the function is evaluated.

SEE ALSO Frame Phrase, FRAME-COL Function, FRAME-LINE function, FRAME-ROW function

FRAME-FIELD Function

During a data entry statement, returns the name of the input field the cursor is in. At other times, returns the name of the input field the cursor was last in.

The FRAME-FIELD function is particularly useful if you want to provide the user with help that is based on the input field they are using. In that case, you can use the FRAME-FIELD function to determine what help information to display when the user presses HELP (F2) to run the applhelp.p procedure you have written. For more information on writing a help procedure, see Chapter 3 of the *Programming Handbook*.

SYNTAX

```
FRAME-FIELD
```

EXAMPLE

```
r-frfld.p

FOR EACH customer:
  UPDATE cust-num name address address2 city
      st zip WITH 1 DOWN 1 COLUMN CENTERED
      EDITING:
        DISPLAY
        "You are editing field:" FRAME-FIELD SKIP
        "Of file:" FRAME-FILE SKIP
        "Of database:" FRAME-DB SKIP
        "Its value is:" FRAME-VALUE FORMAT "x(20)"
        WITH FRAME a ROW 15 NO-LABELS CENTERED.
        READKEY.
        APPLY LASTKEY.
      END. /* Editing */
  END.
```

For each field the user is updating, this procedure displays the name of the field, the file the field belongs to, and the value currently in the field. The EDITING phrase is part of the UPDATE statement; it displays information about the field as you update the record, and then reads each of the keystrokes entered (READKEY) and applies those keystrokes (APPLY LASTKEY).

NOTES

- If the current or last input field is an array, FRAME-FIELD returns the name of the field but does not indicate the array element that the input field represents. To display the array element, use the FRAME-INDEX function.

- If the cursor was not in an enabled input field when the last input statement ended, FRAME-FIELD returns an empty string.
- The FRAME-FIELD value is set to blanks at the next pause (whether done by a PAUSE statement or automatically by PROGRESS) or at the next READKEY statement.

SEE ALSO FRAME-FILE Function, FRAME-INDEX, FRAME-VALUE Function, PROGRAM-NAME Function, Chapter 3 of the *Programming Handbook*.

FRAME-FILE Function

Returns the name of the file containing the field the cursor is in. The FRAME-FILE function is useful if you want to provide users with context-sensitive help. For more information on writing help procedures, see Chapter 3 of the *Programming Handbook.*

SYNTAX

```
FRAME-FILE
```

EXAMPLE

r-frfile.p

```
FOR EACH customer, EACH order OF customer:
  DISPLAY order-num WITH CENTERED ROW 2 FRAME onum.
  UPDATE
  customer.cust-num AT 5 order.cust-num AT 30 SKIP
  customer.name AT 5 order.name AT 30 SKIP
  customer.city AT 5 order.city AT 30 SKIP
  customer.st AT 5 order.st AT 30 SKIP
  customer.zip AT 5 order.zip AT 30

  WITH ROW 8 CENTERED 1 DOWN NO-LABELS
  MESSAGE "      The field" FRAME-FIELD
->           "is from the" FRAME-FILE "file    ".
  READKEY.
  APPLY LASTKEY.
  END. /*Editing */
END.
```

This procedure updates fields from both the order file and the customer file. It uses the FRAME-FILE function to tell you which file the field being updated is in.

NOTES

- FRAME-FILE returns a null string if the frame field being entered is a variable.
- If the cursor is not in an enabled input field when the last input statement ends, FRAME-FILE returns a null string.
- The FRAME-FILE value is set to blanks at the next PAUSE statement, at the next READKEY statement, or when PROGRESS pauses or automatically.

SEE ALSO FRAME-FIELD Function, FRAME-VALUE Function, PROGRAM-NAME Function, Chapter 3 of the *Programming Handbook*

FRAME-INDEX Function

During a data entry statement, returns the subscript of the array element of the input field to which the cursor is currently positioned. At other times, returns the subscript of the array element to which the cursor was last positioned.

The FRAME-INDEX function is particularly useful if you want to provide the user with help for the input array element being edited. To use the FRAME-INDEX function for that purpose, you must determine what help information to display when the user presses HELP (F2). For more information on writing a help procedure, see Chapter 3 of the *Programming Handbook.*

SYNTAX

```
FRAME-INDEX
```

EXAMPLE

r-frindx.p

```
DEFINE VARIABLE menu AS CHARACTER EXTENT 3.
DO WHILE TRUE:
    DISPLAY
      "1. Display Customer Data" @ menu[1] SKIP
      "2. Display Order Data"    @ menu[2] SKIP
      "3. Exit"                  @ menu[3] SKIP
      WITH FRAME choices NO-LABELS.
    CHOOSE FIELD menu AUTO-RETURN WITH FRAME choices
      TITLE "Demonstration Menu" WITH CENTERED ROW 10.
    HIDE FRAME choices.
    IF FRAME-INDEX EQ 1 THEN
         MESSAGE "You picked option 1.".
    ELSE IF FRAME-INDEX EQ 2 THEN
         MESSAGE "You picked option 2.".
    ELSE IF FRAME-INDEX EQ 3 THEN LEAVE.
END.
```

In this example the FRAME-INDEX function uses the cursor position to determine which option you have chosen.

NOTES

- If the cursor is not in an enabled input field when the last input statement is executed, FRAME-INDEX returns a 0.
- The FRAME-INDEX value is set to 0 at the next pause (whether done by a PAUSE statement or automatically by PROGRESS) or at the next READKEY statement.

SEE ALSO Frame Phrase, FRAME-DB Function, FRAME-FIELD Function, FRAME-FILE Function.

FRAME-LINE Function

Returns an integer value that represents the current logical line number in a down frame.

SYNTAX

```
FRAME-LINE [ (frame) ]
```

(frame)

The name of the frame in which you are trying to determine a line number. If you do not supply a frame name, the FRAME-LINE function uses the default frame for the block containing the FRAME-LINE function. If the FRAME-LINE function is in a DO block, the function uses the default frame scoped to the block containing the DO block.

EXAMPLE

r-frline.p

```
DEFINE VARIABLE ans AS LOGICAL
       LABEL "Do you want to delete this customer ?".

STATUS INPUT "Enter data, or use the " + KBLABEL("get")
             + " key to delete the customer".
get-cust:
  FOR EACH customer WITH 10 DOWN:
    UPDATE cust-num name max-credit
    editing:
      READKEY.
      IF KEYFUNCTION(lastkey) = "get"
      THEN DO:
➤       UPDATE ans WITH ROW FRAME-ROW + 3 + FRAME-LINE + 5
          COLUMN 10 SIDE-LABELS OVERLAY FRAME del-frame.
        IF ans
        THEN DO:
          DELETE customer.
          NEXT get-cust.
        END.
      END.
      APPLY LASTKEY.
    END.
END.
```

This procedure lists customers and allows the user to delete customers one at a time. When the user presses F5 to delete a customer, the procedure displays an overlay frame below the last customer displayed. The overlay frame asks "Do you want to delete this customer?" The user answers yes or no. The position of the overlay frame is calculated from the upper right corner of the frame and the current line within the frame. That is, FRAME-ROW + 3 + FRAME-LINE gives the position of the current line in the frame, taking into account the three lines for the frame box and the labels. The prompt is placed five lines below the current line.

NOTES

- If there is a down pending for a frame, the FRAME-LINE function returns a value equal to FRAME-LINE + 1.
- The FRAME-LINE function counts an underline row as a logical line. A logical line corresponds to one iteration in a down frame and may contain more than one physical line.
- The FRAME-LINE function returns a value of 0 if the frame is not in view when the function is evaluated.

SEE ALSO Frame Phrase, FRAME-COL Function, FRAME-DOWN Function, FRAME-ROW Function

FRAME-NAME Function

Returns the name of the frame, if the cursor was last positioned to a field that is enabled for input.

SYNTAX

```
FRAME-NAME
```

EXAMPLE

r-frname.p

```
FOR EACH customer, EACH order OF customer:
  DISPLAY order-num WITH CENTERED ROW 2 FRAME onum.
  UPDATE
    customer.cust-num AT 5 customer.name AT 30 SKIP
      WITH FRAME custfrm WITH CENTERED 1 DOWN
    EDITING:
      DISPLAY "You are currently editing a frame called"
➤        FRAME-NAME WITH FRAME d1 WITH 1 DOWN CENTERED.
     READKEY.
     APPLY LASTKEY.
     IF LASTKEY = KEYCODE("RETURN") THEN
      MESSAGE "Press the space bar to edit order shipdate".
    END.  /* Editing */
  HIDE FRAME custfrm.
  HIDE FRAME d1.
  UPDATE
    sdate AT 5
     WITH FRAME orderfrm WITH CENTERED 1 DOWN
   EDITING:
    DISPLAY "Now you are editing a frame called"
➤       FRAME-NAME WITH FRAME d2 WITH 1 DOWN CENTERED.
    READKEY.
    APPLY LASTKEY.
   END.
  HIDE FRAME orderfrm.
  HIDE FRAME d2.
END.
```

This procedure displays customer information in one frame, then displays order information for the customer in a second frame. The FRAME-NAME function is used to display the name of the frame in which the cursor is currently positioned.

NOTES

- The FRAME-NAME function returns an empty string for a frame that has not been named (the default frame). It also returns an empty string if the cursor was last positioned to a field that is not enabled for input.
- When using the FRAME-NAME function, you must place it logically following the frame phrase in which it is named.
- FRAME-NAME is especially useful for context-sensitive help.

SEE ALSO Frame Phrase, PROGRAM-NAME Function

FRAME-ROW Function

Returns an integer value that represents the row position of the upper left corner of a frame.

SYNTAX

```
FRAME-ROW [ (frame ) ]
```

(frame)
The name of the frame whose row position you are trying to determine. If you do not supply a frame name, the FRAME-ROW function uses the default frame for the block containing the FRAME-ROW function. If the FRAME-ROW function is in a DO block, the function uses the default frame scoped to the block containing the DO block.

EXAMPLE

```
r-frrow.p

FOR EACH customer:
  DISPLAY customer WITH FRAME cust-frame 2 COLUMNS
          TITLE "CUSTOMER INFORMATION".
  FOR EACH order OF customer:
    DISPLAY order-num odate sdate pdate shp-via misc-info
            cust-po shipped WITH 2 COLUMNS 1 DOWN OVERLAY
            TITLE "CUSTOMER'S ORDERS"
  ➤         ROW FRAME-ROW(cust-frame) + 8
            COLUMN FRAME-COL(cust-frame) + 1.
  END.
END.
```

This procedure displays customer information in one frame, then displays order information for the customer in a second frame that overlays the first. FRAME-ROW and FRAME-COL control the placement of the overlay frame. FRAME-ROW places the overlay frame on the eighth row of the first frame. FRAME-COL places the overlay frame on the first column of the first frame.

SEE ALSO Frame Phrase, FRAME COL-Function, FRAME-DOWN Function, FRAME-LINE Function

FRAME-VALUE Function

During a data entry statement, returns the (character string) value of the input field the cursor is in. At other times, returns the (character string) value of the input field the cursor was last in.

SYNTAX

```
FRAME-VALUE
```

EXAMPLE

r-frval.p

```
FOR EACH customer:
  DISPLAY cust-num name address city st zip
          WITH 1 COLUMN 1 DOWN COLUMN 20.
  SET address city st zip.
END.

DISPLAY
  "You were updating field:" FRAME-FIELD
    FORMAT "x(20)" SKIP
  "Of file:" FRAME-FILE SKIP
→ "And it had the value:" FRAME-VALUE
    FORMAT "x(20)" SKIP(2)
   "When you pressed the END key"
   WITH FRAME helper NO-LABELS COLUMN 20
        ROW 14 NO-BOX.
```

When the user presses END-ERROR (F4) while running this procedure, the procedure displays the name and value of the field the user was updating, along with the name of the file containing that field.

NOTES

- If the cursor is not in an enabled input field when the last input statement ends, FRAME-VALUE returns a null string.
- FRAME-VALUE is set to blanks at the next pause (either done by a PAUSE statement or automatically by PROGRESS) or at the next READKEY statement.
- FRAME-VALUE returns strings. If you use FRAME-VALUE to return a number, you must convert it prior to numeric comparisons. For example:
  `FIND customer WHERE cust-num=INTEGER(FRAME-VALUE).`

SEE ALSO FRAME-FIELD Function, FRAME-FILE Function, PROGRAM-NAME Function, FRAME-VALUE Statement, Chapter 3 of the *Programming Handbook*

FRAME-VALUE Statement

During a data entry statement, stores the value of an expression in a frame field.

SYNTAX

```
FRAME-VALUE = expression
```

expression

A constant, field name, variable name or any combination of these whose value you want to store in a frame field. If there is no frame active when PROGRESS runs this statement, you receive an error message. Otherwise, if the frame is in view, PROGRESS redisplays the field.

The data type of the *expression* must be the same as the data type of the frame field in which you are storing that expression. However, if the data type of *expression* is character, PROGRESS will store characters in the frame field regardless of the data type of that frame field, truncating characters if necessary.

The FRAME-VALUE statement can pass information from an applhelp.p procedure to the calling procedure. For example, if the user enters a value into a field called help-field, pass that value back to the calling procedure with the statement:

```
FRAME-VALUE = INPUT help-field.
```

EXAMPLE

r-frmval.p

```
DEFINE VARIABLE txt AS CHARACTER INITIAL "PROGRESS".
DEFINE VARIABLE tmpdate AS DATE INITIAL TODAY.

STATUS INPUT "Enter data or use the "
             + KBLABEL("get")
             + " key to enter the unknown value (?)".
UPDATE txt tmpdate
EDITING:
  READKEY.
  IF KEYFUNCTION(LASTKEY) = "GET"
  THEN DO:
    FRAME-VALUE = ?.
    NEXT.
  END.
  APPLY LASTKEY.
END.
```

This procedure displays the word PROGRESS, the date, and "Enter data or use the F5 key to enter the unknown value (?)". You can update the information in the frame. If you press F5, the r-frmval.p procedure assigns the unknown value to a field with the FRAME-VALUE statement.

SEE ALSO FRAME-FIELD Function, FRAME-FILE Function, FRAME-VALUE Function, Chapter 7 of the *Programming Handbook*

GATEWAYS Function

The GATEWAYS function takes no arguments. It returns a string containing a list of database types supported by the PROGRESS product from which it is executed. For example:

“PROGRESS,ORACLE,RMS”

The returned string is a comma-separated list of strings that you can use with the LOOKUP function. Notice that the string includes PROGRESS database names and non-PROGRESS database names. For example, the string shown above contains PROGRESS and ORACLE.

SYNTAX

```
GATEWAYS
```

EXAMPLE

r-gatewy.p

```
IF LOOKUP("ORACLE",GATEWAYS) > 0 AND DBVERSION(1) = "6"
  THEN MESSAGE "ORACLE Version 6 is supported.".
ELSE IF LOOKUP("ORACLE",GATEWAYS) > 0 AND DBVERSION(1) = "5"
  THEN MESSAGE "ORACLE Version 5 is supported.".
```

This procedure displays a message if ORACLE Version 5 or ORACLE Version 6 is supported by the current PROGRESS product.

SEE ALSO CONNECT, DISCONNECT, CREATE ALIAS, and DELETE ALIAS statements; CONNECTED, LDBNAME, PDBNAME, DBTYPE, DBRESTRICTIONS, FRAME-DB, and NUM-DBS functions.

GE or > = Operator

Returns a TRUE value if the first of two expressions is greater than or equal to the second expression.

SYNTAX

$$\textit{expression} \left\{ \begin{matrix} \text{GE} \\ >= \end{matrix} \right\} \textit{expression}$$

expression

A constant, field name, variable name, or any combination of these. The expressions on either side of the GE or > = must be of the same data type, although one may be integer and the other decimal.

EXAMPLE

```
                                                    ge.p
→ FOR EACH item WHERE on-hand >= 120:
    DISPLAY item.item-num idesc on-hand.
  END.
```

This procedure displays item information for those items whose on-hand value is greater than or equal to 120.

NOTES

- If one of the expressions has an unknown value and the other does not, the result is FALSE. The exception to this is SQL. For SQL, if either or both expressions is unknown, then the result is unknown.

  You can compare character strings with GE. Most character comparisons are case-insensitive in PROGRESS. That is, all characters are converted to upper case prior to comparisons. However, it is possible to define fields and variables as case-sensitive (though it is not advised, unless strict ANSI SQL adherence is required). If either *expression* is a field or variable defined to be case-sensitive, the comparison is case-sensitive and "Smith" does not equal "smith."

  Note that because characters are converted to their ASCII values for comparison, all upper case letters sort before all lower case letters (**a** is greater than **Z**, but less than **b**). Note also that in ASCII uppercase **A** is less than " [", " \ ", " ^ ", "_", and " ' " but lowercase **a** is greater than these.

GETBYTE

Returns the integer value of the specified byte

SYNTAX

```
GETBYTE(raw expression,position)
```

raw expression
A function or variable name that returns a raw data.

position
An integer value that indicates the position of the byte that you want to find the integer value of.

EXAMPLE

rawget.p

```
/*You must connect to a non-PROGRESS demo database to
run this procedure*/

 DEFINE VAR i AS INT.

 FOR EACH customer:
        i = GETBYTE(RAW(name),1).
        IF i = 83 THEN DISPLAY NAME.
 END.
```

In this example, the RAW Function goes to the customer field in the non-PROGRESS database and retreives the first byte. The GETBYTE function then stores the integer value of that byte in the variable i. The procedure then tests the value, if the integer value is 83 (the ASCII value for S) PROGRESS displays the name.

NOTES

- PROGRESS returns a -1 if *n* is greater than the length of *expression* or if *n* is less than one.
- If *raw expression* is the unknown value, GETBYTE returns the unknown value.

SEE ALSO LENGTH, RAW Function, RAW Statement, PUTBYTE

GO-PENDING Function

Returns a TRUE value if, within an EDITING phrase, an APPLY statement results in a GO action. The GO action is deferred until the end of the EDITING phrase.

SYNTAX

```
GO-PENDING
```

EXAMPLE

r-gopend.p

```
REPEAT:
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num.
  UPDATE
    name address city st SKIP
    max-credit curr-bal WITH 1 COLUMN
    EDITING:
      /* Read and apply the last key pressed */
      READKEY.
      APPLY LASTKEY.
      /* If the user pressed GO and the current
         balance is greater than the max-credit,
         display a message and do the next
         iteration of the EDITING phrase */
➤     IF GO-PENDING AND
        INPUT curr-bal > INPUT max-credit
        THEN DO:
          MESSAGE
          "Current balance exceeds credit limit".
          NEXT.
        END.
    END.
END.
```

The r-gopend.p procedure lets you update some of the fields in each customer record. If you press GO (F1) when the value in the current balance field is greater than the balance in the max-credit field, the UPDATE statement does not end. Instead, it continues prompting for input until you correct the problem and then press GO (F1).

SEE ALSO APPLY Statement, EDITING Phrase

GRANT Statement (SQL)

Allows the owner or any user who holds the GRANT OPTION on a table or view to grant privileges on that table or view.

SYNTAX

```
GRANT
 { ALL [ PRIVILEGES ] |
   { SELECT |
     INSERT |
     DELETE |
    { UPDATE [(column-list)] }        } [,...] }
 ON table-name TO { grantee-list | PUBLIC } [ WITH GRANT OPTION ]
```

ALL [PRIVILEGES]
: Grants all privileges that the granting user has (SELECT, INSERT, DELETE, and UPDATE) to the specified users. If the user granting the privileges is lacking one or more of these, they are not passed on.

SELECT
: Grants the SELECT (_can-read) privilege to the specified users.

INSERT
: Grants the INSERT (_can-create) privilege to the specified users.

DELETE
: Grants the DELETE (_can-delete) privilege to the specified users.

UPDATE [*(column-list)*]
: Grants the UPDATE (_can-write) privilege to the specified users. You can list the columns that the user can update. If you specify the keyword UPDATE but omit the column list, the grantees can update all columns of the specified table.

ON *table-name*
: The name of the table or view on which you want to grant privileges.

TO {*grantee-list* | PUBLIC}
: The users to whom you want to grant privileges. You can specify either a list of user names or the keyword PUBLIC which grants the privileges to all users.

[WITH GRANT OPTION]

Grants the recipient of the privileges the right to, in turn, grant those privileges to other users, and to revoke the privileges from anyone except the owner of the table. If this option is omitted, the recipient cannot grant the privileges to other users.

EXAMPLES

```
GRANT ALL PRIVILEGES
    ON employee
    TO nancy
    WITH GRANT OPTION.
```

```
GRANT SELECT
    ON doc
    TO PUBLIC.
```

```
GRANT SELECT, INSERT, UPDATE
    ON employee
    TO alan, bob, kathy.
```

NOTE

- The GRANT statement can be used only in interactive SQL.

GT or > Operator

Returns a TRUE value if the first of two expressions is greater than the second expression.

SYNTAX

expression $\left\{ \begin{matrix} \text{GT} \\ > \end{matrix} \right\}$ *expression*

expression

A constant, field name, variable name, or any combination of these. The expressions on either side of the GT or > must be of the same data type, although one may be integer and the other decimal.

EXAMPLE

gt.p

```
FOR EACH item:
  IF alloc > 0
  THEN IF (on-hand <= 0) OR
    (alloc / on-hand > .9)
  THEN DISPLAY item-num idesc  on-hand alloc.
END.
```

This procedure lists all items which have a negative on-hand quantity or more than 90% of the on-hand inventory currently allocated.

NOTES:

- If one of the expressions has an unknown value and the other does not, the result is FALSE. The exception to this is SQL. For SQL, if either or both expressions is unknown, then the result is unknown.

- You can compare character strings with GT. Most character comparisons are case-insensitive in PROGRESS. That is, all characters are converted to upper case prior to comparisons. However, it is possible to define fields and variables as case-sensitive (though it is not advised, unless strict ANSI SQL adherence is required). If either *expression* is a field or variable defined to be case-sensitive, the comparison is case-sensitive and "Smith" does not equal "smith."

  Note that because characters are converted to their ASCII values for comparison, all upper case letters sort before all lower case letters (**a** is greater than **Z**, but less than **b**).

HIDE Statement

Removes a frame from the terminal screen, or clears the message area, or clears all frames and messages. If the frame is a PAGE-TOP or PAGE-BOTTOM frame, then it is removed from the list of active frames for printing at the top or bottom of each page.

SYNTAX

```
                             [ FRAME frame ]
HIDE [STREAM stream ]        [ MESSAGE     ]     [ NO-PAUSE ]
                             [ ALL         ]
```

STREAM *stream*

Specifies the name of a stream. If you do not name a stream, the unnamed stream is used. See the DEFINE STREAM reference page and Chapter 9 of the *Programming Handbook* for more information about streams.

FRAME *frame*

The name of the frame you want to hide. If you do not use this option or the MESSAGE or ALL options, HIDE hides the default frame for the block containing the HIDE statement.

MESSAGE

Hides all messages displayed in the message area. If you use the PUT SCREEN statement to display data in the message area, the HIDE MESSAGE statement does not necessarily hide that data.

ALL

Hides all frames and the message area.

NO-PAUSE

Does not pause before hiding. Ordinarily, if data has been displayed but there have been no data entry operations or pauses, PROGRESS asks you to "Press space bar to continue" before hiding the frame.

EXAMPLE

r-hide.p

```
DEFINE VARIABLE selection AS INTEGER FORMAT "9".
FORM   "Please Make A Selection:" SKIP(2)
       "     1. Hide Frame A.     " SKIP
       "     2. Hide Frame B.     " SKIP
       "     3. Hide All.         " SKIP
       "     4. Hide This Frame   " SKIP
       "     5. Exit              " SKIP(2)
       WITH FRAME x NO-LABELS.

REPEAT:
  VIEW FRAME x.
  DISPLAY "This is Frame A."
    WITH FRAME a ROW 1 COLUMN 60.
  DISPLAY "This is Frame B."
    WITH FRAME b ROW 16 COLUMN 10 4 DOWN.
  MESSAGE "Make your selection!".
  UPDATE "Selection: " selection
    VALIDATE(0 < selection AND selection < 7,
             "Invalid selection") AUTO-RETURN
    WITH FRAME x.
  IF selection = 1 THEN HIDE FRAME a.
  ELSE IF selection = 2 THEN HIDE FRAME b.
  ELSE IF selection = 3 THEN HIDE ALL.
  ELSE IF selection = 4 THEN HIDE FRAME x.
  ELSE IF selection = 5 THEN LEAVE.
  PAUSE.
END.
```

This procedure uses the HIDE statement to hide various frames. As the procedure runs, the frames come back into view when the DISPLAY statements are executed in each iteration.

NOTES

- When a block iterates, any display frame that is scoped to the block or to a nested block is tagged for hiding (unless you have used the NO-HIDE option in the Frame phrase), but is not hidden yet. Then, the first frame activity of the next iteration (a DISPLAY, INSERT, PROMPT-FOR, SET, VIEW, or UPDATE statement) for a frame scoped to the block or to a nested block causes all tagged frames to be hidden.

The frame associated with that first frame activity is not hidden because it would be redisplayed immediately. By not hiding and then redisplaying this frame, screen displays are considerably faster. When a block ends, all frames scoped to that block or to nested blocks have their "hide" tags removed.

- Frames displayed by procedures within a block or within a nested block are treated the same way as other frames in a nested block.
- When a frame is to be displayed and there is not enough room on the screen, PROGRESS automatically hides one or more frames. Frames are hidden one at a time starting with the lowest frame on the screen until there is room to fit the new frame.
- It is more efficient to HIDE ALL than to individually HIDE every frame in view.
- If you are working with a PAGE-TOP or PAGE-BOTTOM frame, you use the VIEW or DISPLAY statement to activate that frame. The VIEW statement does not actually bring a PAGE-TOP or PAGE-BOTTOM frame into view. It simply activates the frame so that when a new page begins or ends, PROGRESS displays the frame. If you use the HIDE statement to hide a PAGE-TOP or PAGE-BOTTOM frame, PROGRESS deactivates that frame so that it can no longer be brought into view unless reactivated with a VIEW or DISPLAY statement.
- If output is not directed to the terminal, HIDE has no effect on the terminal screen.

SEE ALSO VIEW Statement

IF...THEN...ELSE Function

Evaluates one of two expressions depending on the value of a specified condition.

SYNTAX

```
IF condition  THEN expression1     ELSE expression2
```

condition

A logical expression which is a constant, field name, variable name, or any combination of these whose value is logical (TRUE/FALSE).

expression1

A constant, field name, variable name, or any combination of these.

expression2

A constant, field name, variable name, or any combination of these whose value is of a data type that is compatible with the data type of *expression1*.

EXAMPLE

r-ifelsf.p

```
FOR EACH customer
  BY IF sales-region = "East" THEN 1
  ELSE (IF sales-region = "Central" THEN 2 ELSE 3):
  DISPLAY sales-region name sales-rep.
END.
```

You can use the IF...THEN...ELSE function when you want to sort records in an unusual order. In this example, the customers are sorted based on sales-region in the order East, then Central, then all others (e.g. West). If the first statement of the procedure were FOR EACH customer BY sales-region, the order would be Central, East, West.

IF...THEN...ELSE Statement

Makes execution of a statement or block of statements conditional. If the value of the expression following the IF statement is TRUE, PROGRESS processes the statements following the THEN statement. Otherwise, PROGRESS processes the statements following the ELSE statement.

SYNTAX

$$\text{IF } \textit{expression} \quad \text{THEN}\left\{ \begin{matrix} \textit{block} \\ \textit{statement} \end{matrix} \right\} \left[\text{ELSE}\left\{ \begin{matrix} \textit{block} \\ \textit{statement} \end{matrix} \right\} \right]$$

expression

A constant, field name, variable name, or any combination of these whose value is TRUE or FALSE. The expression may include comparisons, logical operators, and parentheses.

THEN

Describes the processing to do if the *expression* is TRUE.

block

The block whose processing you want to do if *expression* is TRUE. See the DO, FOR EACH, or REPEAT reference pages for more information. If you do not start a block, you can process just one statement after both the IF keyword or the ELSE keyword.

Any block or blocks you use in an IF...THEN...ELSE statement can contain other blocks or other IF...THEN...ELSE statements.

statement

A single PROGRESS statement. The *statement* may be another IF...THEN...ELSE statement. If you want to use more than one statement, you must enclose those statements in a DO, FOR EACH, or REPEAT block.

ELSE

Describes the processing to do if the *expression* is FALSE. You do not have to use the ELSE option.

EXAMPLE

r-ifelss.p

```
DEFINE VARIABLE ans AS LOGICAL.
DEFINE STREAM due.

OUTPUT STREAM due TO ovrdue.lst.
DISPLAY STREAM due
  "Orders shipped but still unpaid as of" TODAY SKIP(2)
  WITH NO-BOX NO-LABELS CENTERED FRAME hdr PAGE-TOP.

FOR EACH order WITH FRAME oinfo:
  DISPLAY order-num name odate pdate sdate.
➤ IF sdate = ? THEN DO:
➤   IF pdate = ? THEN DO:
      MESSAGE "Please update the promise date."
      REPEAT WHILE pdate = ?:
        UPDATE pdate WITH FRAME oinfo.
      END.
    END.
    ans = false.
    MESSAGE "Has this order been shipped?" UPDATE ans.
    IF ans THEN REPEAT WHILE sdate = ?:
      UPDATE sdate WITH FRAME oinfo.
    END.
  END.
  ELSE DO:
    ans = true.
    MESSAGE "Has this order been paid?" UPDATE ans.
➤   IF NOT ans THEN DO:
      DISPLAY STREAM due order-num TO 14 name AT 18
                         odate AT 42 sdate AT 54
      WITH NO-BOX DOWN FRAME unpaid.
    END.
  END.
END.
OUTPUT STREAM due CLOSE.
```

The r-ifelss.p procedure creates a report in a file listing customers whose orders have been shipped but who have not paid for those orders.

First, the procedure writes report headers to the ovrdue.lst file. Next, the outer FOR EACH block reads each of the orders using a DISPLAY statement to display information about each order. If there are no values in the sdate and pdate fields, the procedures asks that you enter a promise date. The procedure then asks if the order has been shipped. If so, you must supply a ship date.

If there is a ship date and a promise date for an order, the procedure asks if the order has been paid for. If not, the procedure displays the order information to the file.

IMPORT Statement

The IMPORT Statement is the counterpart of the EXPORT statement. It reads a line from an input file that, typically, has been created by EXPORT.

SYNTAX

```
IMPORT [STREAM stream] { fields                           }
                       { record [EXCEPT field ...]        }
                       { [ ^ ]                            }
```

STREAM *stream*
: Specifies the name of a stream. If you do not name a stream, the unnamed stream is used.

fields
: One or more space-separated field or variable names to which you are importing data. Use the caret " ^ " to skip a data value in each input line when input is being read from a file. If an input data line contains an unquoted hyphen in place of a data value, then the corresponding field is skipped, as in UPDATE.

record
: The name of a record buffer. All of the fields in the record are processed exactly as if you had named each of them individually. The record you name must contain at least one field. To use IMPORT with a record in a file defined for multiple databases, you must qualify the record's filename with the database name. See the description of the Record Phrase for more information.

EXCEPT *field*
: All fields except those fields listed in the EXCEPT phrase will be imported.

EXAMPLE

```
r-imprt.p

INPUT FROM customer.d.
REPEAT:
  CREATE customer.
  IMPORT customer.
END.
```

This procedure takes the data in file `customer.d`. and enters it into PROGRESS database file `customer`.

NOTES

- Commonly, IMPORT is used following an INPUT FROM statement.
- The statement,

  ```
  IMPORT ^
  ```

  reads a line from an import file and ignores that line. This is an efficient way of skipping lines.
- The data in the input stream must be in a *standard format* to be read back into PROGRESS. All character fields must be enclosed in quotes (" ") if they contain any blanks or break characters. Any quotes contained in the data to be imported must be replaced by two quotes ("" ""). An unknown value must be displayed as an unquoted question mark (?).
- Data read in with IMPORT is not restricted by frame related format statements, as is data read in by SET or UPDATE. Since IMPORT does not have to validate the input stream, it is faster than SET or UPDATE.
- You can also access the IMPORT statement from the Admin menu in the PROGRESS Data Dictionary.
- EXPORT is sensitive to the Date format (-d), Century (-yy), and European numeric (-E) startup options. When loading data with the IMPORT statement, use the same settings that you used with the EXPORT statement.

SEE ALSO EXPORT Statement, DEFINE STREAM Statement, DISPLAY Statement, INPUT FROM Statement, PUT Statement, and STRING Function.

INDEX Function

Returns an integer indicating the position of the target string within the source string.

SYNTAX

INDEX(*source,target*)

source
: A character expression (a constant, field name, variable name, or any combination of these that results in a character value).

target
: A character expression whose position you want to locate in *source*. If *target* does not exist within *source*, INDEX returns 0.

EXAMPLE

r-index.p

```
DEFINE VARIABLE x AS CHARACTER FORMAT "9"
  LABEL "Enter a digit between 1 and 5".
DEFINE VARIABLE show AS CHARACTER FORMAT "x(5)"
  EXTENT 5 LABEL "Literal".
show[1] = "One".
show[2] = "Two".
show[3] = "Three".
show[4] = "Four".
show[5] = "Five".
REPEAT:
  SET x AUTO-RETURN.
  IF INDEX("12345", x) = 0 THEN DO:
    MESSAGE "x must be 1,2,3,4, or 5. Try again.".
    UNDO, RETRY.
  END.
  ELSE DISPLAY show[INTEGER(x)].
END.
```

For this example, you must enter 1,2,3,4, or 5. The INDEX function checks to see if the digit exists in the string "12345".

NOTES

- If either operand is case-sensitive, then the search is case-sensitive.
- If the target string is null, the result is 0.

SEE ALSO LOOKUP Function, R-INDEX Function

INPUT Function

References the value of a field in a screen buffer (frame). For example, if you use the PROMPT-FOR statement to get input from the user, PROMPT-FOR stores that information in the screen buffer. You can use the INPUT function to refer to that information.

SYNTAX

```
INPUT [ FRAME frame ] field
```

FRAME *frame*

The name of the frame containing the field named by the *field* argument. If you do not name a frame, the INPUT function starts with the current frame and searches outward until it finds the field you name with the *field* argument.

field

The name of a field or variable whose value is stored in the screen buffer.

EXAMPLE

r-input.p

```
FOR EACH customer:
  DISPLAY cust-num name max-credit LABEL "Current max credit"
          WITH FRAME a 1 DOWN ROW 1.
  PROMPT-FOR max-credit LABEL "New max credit"
             WITH SIDE-LABELS NO-BOX ROW 10 FRAME b.
  IF INPUT FRAME b max-credit <> max-credit
  THEN DO:
    DISPLAY "Changing max credit of" name SKIP
            "from" max-credit "to" INPUT FRAME b max-credit
            WITH FRAME c ROW 15 NO-LABELS.
→   max-credit = INPUT FRAME b max-credit.
  END.
  ELSE DISPLAY "No change in max credit" WITH FRAME d ROW 15.
END.
```

In this procedure, the current max-credit for a customer is displayed. The PROMPT-FOR statement prompts the user for a new max-credit value and stores the supplied data in the screen buffer. The procedure uses the INPUT function to point to the data in that buffer.

If the user enters a new value, the procedure displays a message saying that the value has been changed. If the user enters the same value, the procedure tells the user the max-credit has not been changed.

NOTES

- If a field or variable referenced with INPUT is used in more than one frame, then the value in the frame most recently introduced in the procedure is used. To be sure you are using the appropriate frame, use the FRAME option with the INPUT function to reference a particular frame.
- If you use the INPUT function for a character field whose format contains fill characters, then the value of the function does not contain the fill characters. The fill characters are not stored in the database field or variable, but are instead supplied during display formatting of the data.

INPUT CLEAR Statement

Clears any keystrokes buffered from the keyboard, discarding any "type-ahead" characters. The INPUT CLEAR statement is particularly useful when you want to be sure that if the user types more characters than are required for a field, those characters are not used in the input statement that follows it.

SYNTAX

```
INPUT CLEAR
```

EXAMPLE

r-inclr.p

```
    DISPLAY "     Please choose      " SKIP
            " 1 Run order entry      " SKIP
            " 2 Run receivables      " SKIP
            " 3 Exit                 "
            WITH CENTERED FRAME menu.

    REPEAT:
      READKEY.
      IF LASTKEY = KEYCODE("1") THEN RUN ordentry.
      ELSE
      IF LASTKEY = KEYCODE("2") THEN RUN receive.
      ELSE
      IF LASTKEY = KEYCODE("3") THEN QUIT.
      ELSE DO:
        MESSAGE "Sorry, that is not a valid choice".
-->     INPUT CLEAR.
      END.
    END.
```

This menu procedure tests each key the user presses. If the user presses a key other than 1, 2, or 3, the keyboard buffer is cleared and a message displayed.

NOTES

- Under DOS and OS/2, the keyboard type-ahead buffer contains a maximum of 16 characters.
- If the current input source is not the keyboard, the INPUT CLEAR statement has no effect.

INPUT CLOSE Statement

Closes the default input source or the stream you name.

SYNTAX

```
INPUT [ STREAM stream ] CLOSE
```

STREAM *stream*

Specifies the name of a stream. If you do not name a stream, the unnamed stream is used. See the DEFINE STREAM reference page and Chapter 9 of the *Programming Handbook* for more information about streams.

EXAMPLE

r-in.p

```
    INPUT FROM VALUE(SEARCH("r-in.dat")).

    REPEAT:
      PROMPT-FOR cust.cust-num tax-no.
      FIND customer USING cust-num.
      ASSIGN tax-no.
    END.
➤   INPUT CLOSE.
```

Instead of getting input from the terminal, this procedure gets input from a file named r-in.dat in the proguide subdirectory. The SEARCH function determines the full path name of this file. Here is what the contents of the r-in.dat file looks like:

```
1 3187926
2 4126721
5 4982155
```

The PROMPT-FOR statement uses the first data item (1) as the cust-num and the second data item (3187926) as the tax-no. The FIND statement finds the customer whose cust-num is 1 and assigns the value of 3187926 as that customer's tax-no. On the next iteration of the REPEAT block, the PROMPT-FOR statement uses the value of 2 as the cust-num, the value of 4126721 as the tax-no, and so on.

The INPUT CLOSE statement closes the input source, resetting it to the terminal. When you run this procedure, you will see data on your screen. That is simply the echo of the data as the procedure is reading it in from the taxno.dat file. If you do not want to see the data on your screen, add the word NO-ECHO to the end of the INPUT FROM statement.

NOTES

- The default input source is the terminal unless the procedure was called by another procedure in which case the default input source is the one that was active in the calling procedure when the second procedure was called.
- When a procedure ends, all input sources established in that procedure are closed.

SEE ALSO DEFINE STREAM Statement, INPUT FROM Statement, Chapter 9 of the *Programming Handbook*

INPUT FROM Statement

Specifies a new input source.

SYNTAX

```
INPUT [ STREAM stream ] FROM

  { opsys-file          }   [ ECHO                  ]
  { opsys-device        }   [ NO-ECHO               ]
  { TERMINAL            }   [ UNBUFFERED            ] . . .
  { VALUE( expression ) }   [ MAP protermcap-entry  ]
                            [ [NO-] MAP             ]
```

STREAM *stream*

Specifies the name of a stream. If you do not name a stream, the unnamed stream is used. See the DEFINE STREAM reference page and Chapter 9 of the *Programming Handbook* for more information about streams.

opsys-file

The name of the file containing the data you want to input to a procedure. If you do not specify a full path name, PROGRESS looks for the file relative to your working directory. Also, remember that UNIX file names are case-sensitive.

opsys-device

The name of a UNIX, DOS, OS/2, VMS, or BTOS/CTOS device.

TERMINAL

Indicates that you want to get input from your terminal. The terminal is the default input source. TERMINAL may not be used with STREAM.

VALUE*(expression)*

An expression (a constant, field name, variable name, or any combination of these) whose value is the source from which you want to input data.

ECHO

Displays all input data on the current output device. Data is echoed by default.

NO-ECHO

Accepts input data without displaying it on the current output device. If you do not use this option, INPUT FROM automatically displays input data on the current output device.

[NO-] MAP *protermcap-entry*

The *protermcap-entry* is an entry from the PROTERMCAP file. Use MAP to read from an input stream that uses different character translation than the current stream. Typically, *protermcap-entry* is a slash-separated combination of a standard device entry and one or more language-specific add-on entries (MAP `laserwriter/french` or MAP `hp2/spanish/italian`, for example). PROGRESS uses the protermcap entries to build a translation table for the stream. Use NO-MAP to make PROGRESS bypass character translation altogether. See also Chapter 2 in the *Programming Handbook* for more information on PROTERMCAP and national language support.

UNBUFFERED

Reads one character at a time from a normally buffered data source, such as a file. You should use the UNBUFFERED option only when the input operations of a UNIX or VMS process invoked with the PROGRESS UNIX or VMS statement may be intermingled with the input being done by the PROGRESS statements that follow the INPUT FROM statement.

EXAMPLE

r-in.p

```
➤ INPUT FROM VALUE(SEARCH("r-in.dat")).
  REPEAT:
    PROMPT-FOR cust.cust-num tax-no.
    FIND customer USING cust-num.
    ASSIGN tax-no.
  END.
  INPUT CLOSE.
```

Instead of getting input from the terminal, this procedure gets input from a file named r-in.dat in the proguide subdirectory. The SEARCH function determines the full path name of this file. Here is what the contents of the r-in.dat file look like:

```
1 3187926
2 4126721
5 4982155
```

The PROMPT-FOR statement uses the first data item (1) as the cust-num and the second data item (3187926) as the tax-no. The FIND statement finds the customer whose cust-num is 1 and assigns the value of 3187926 as that customer's tax-no. On the next iteration of the REPEAT block, the PROMPT-FOR statement uses the value of 2 as the cust-num, the value of 4126721 as the tax-no, and so on.

The INPUT CLOSE statement closes the input source, resetting it to the terminal. When you run this procedure, you see data on your screen. That is simply the echoing of the data as the procedure is reading it in from the taxno.dat file. If you do not want to see the data on your screen, add the word NO-ECHO to the end of the INPUT FROM statement.

NOTES

- After your procedure has finished receiving input, use the INPUT CLOSE statement to close that input source. (The input source is automatically closed at the end of the procedure or when another default input source is opened.)

- If the input source and output destination are both the TERMINAL, then ECHO is always in effect.

- Data is read into the PROGRESS procedure using the INSERT, PROMPT-FOR, SET, or UPDATE statements. The data is placed into the frame fields referenced in these statements, and if ECHO is in effect then the frame is output to the current output destination. If you use the NO-ECHO option, then the frame is not output. If a subsequent DISPLAY statement causes the frame to be displayed then the input data is also displayed if the frame is not yet in view.

- If you are using the PROMPT-FOR, SET, or UPDATE statement to read data from a file and there is a piece of data in each line of the file that you want to disregard, you can use the caret symbol (^) in the PROMPT-FOR, SET, or UPDATE statement. For more information about using this symbol, see the reference page for any of those statements.

- If a line consisting of a single period is read, that is treated as if the END-ERROR (F4) key had been pressed. If the period is in quotes (".") it is treated as an ordinary character.

- If end of file is reached, this is treated as if the ENDKEY had been pressed.

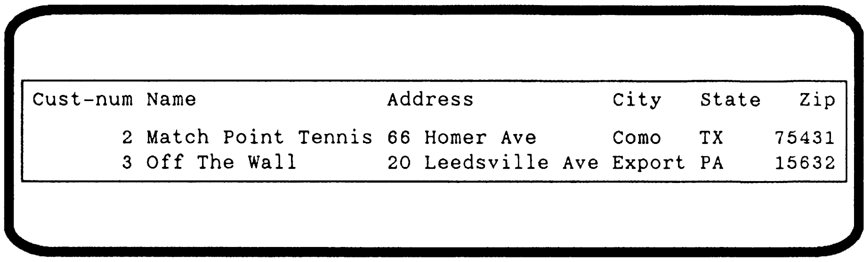
- When you are using the INPUT FROM statement to read data from a file, there are two special characters you can use in that data file: tilde (~) (and \ on UNIX systems) and hyphen (-).

If characters in an input file take up more than one physical line, you can use the tilde (~) character (or the \ character if you are using UNIX) to indicate a line continuation. Here is an input file that uses the tilde character.

```
2 "Match Point Tennis" "66 Homer Ave" "Como"  ~
 "TX" 75431
3 "Off the Wall" "20 Leedsville Ave" "Export" "PA" 15632
```

Do not include a space after the tilde character.

The procedure that uses this input file echos the data as shown here:

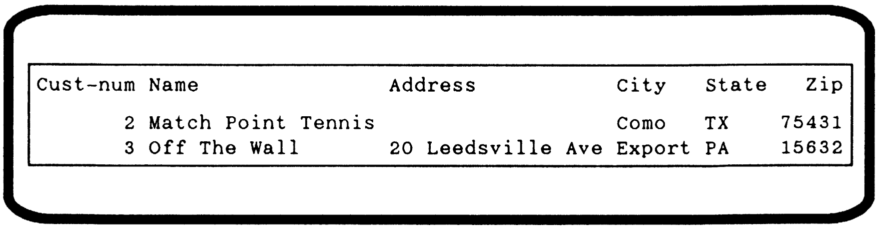


```
Cust-num Name               Address           City   State   Zip

       2 Match Point Tennis 66 Homer Ave      Como   TX    75431
       3 Off The Wall       20 Leedsville Ave Export PA    15632
```

- You can see that the record containing the tilde was treated as a single input line.
- You can use the hyphen in an input file to indicate that you do not want to change the corresponding field in the INSERT, PROMPT-FOR, SET or UPDATE statement. Here is the same input file as shown above, including the hyphen character.

```
2 "Match Point Tennis" - "Como" "TX" 75431
3 "Off The Wall" "20 Leedsville Ave" "Export" "PA" 15632
```

The procedure in this example uses this file to set records in the customer file. When those records are displayed, the address of Match Point Tennis is not changed.



```
Cust-num Name                Address            City    State   Zip
       2 Match Point Tennis                     Como    TX    75431
       3 Off The Wall        20 Leedsville Ave  Export  PA    15632
```

To enter a literal hyphen from a file, enclose it in quotes ("-").

- The quoter program, supplied with PROGRESS, can be used to reformat data files that are not in standard PROGRESS format. For more information about quoter, see Chapter 9 of the *Programming Handbook*.
- If you are using DOS or OS/2, the data in the input file must have the following characteristics:
  - The lines of data in the file are separated by CR-LF pairs.
  - There is no CTRL-Z (EOF) embedded in the file.

SEE ALSO DEFINE STREAM Statement, INPUT CLOSE Statement, INPUT THROUGH Statement, Chapter 9 of the *Programming Handbook*

INPUT THROUGH Statement

Uses the output from a UNIX program as the input to a PROGRESS procedure.

SYNTAX

```
INPUT [ STREAM stream ] THROUGH { program-name       }
                                { VALUE( expression ) }

                                [ argument            ] ...
                                [ VALUE( expression ) ]

                                [ ECHO                  ] ...
                                [ NO-ECHO               ]
                                [ UNBUFFERED            ]
                                [ MAP protermcap-entry  ]
                                [ [NO-] MAP             ]
```

STREAM *stream*

Specifies the name of a stream. If you do not name a stream, the unnamed stream is used. See the DEFINE STREAM reference page and Chapter 9 of the *Programming Handbook* for more information about streams.

program-name

The name of the UNIX program from which you are supplying data to a PROGRESS procedure. This can be a standard UNIX command or your own program.

VALUE*(expression)*

An expression (a constant, field name, variable name, or any combination of these) whose value is the name of a UNIX program from which you are supplying data to a PROGRESS procedure.

argument

An argument you want to pass to the UNIX program. INPUT THROUGH passes this *argument* as a character string.

If the *argument* is the literal value "echo", "no-echo", or "unbuffered" you must enclose it in quotes to prevent PROGRESS from interpreting that argument as one of the ECHO, NO-ECHO, or UNBUFFERED options for the INPUT THROUGH statement.

VALUE*(expression)*

An expression (a constant, field name, variable name, or any combination of these) whose value is an argument you want to pass to the UNIX program. INPUT THROUGH passes the value of *expression* as a character string.

ECHO

Displays all input data on the current output destination. Data is echoed by default.

NO-ECHO

Accepts input data without displaying it on the current output device.

[NO-] MAP *protermcap-entry*

The *protermcap-entry* is an entry from the PROTERMCAP file. Use MAP to read from an input stream that uses different character translation than the current stream. Typically, *protermcap-entry* is a slash-separated combination of a standard device entry and one or more language-specific add-on entries (MAP `laserwriter/french` or MAP `hp2/spanish/italian`, for example). PROGRESS uses the protermcap entries to build a translation table for the stream. Use NO-MAP to make PROGRESS bypass character translation altogether. See also Chapter 2 in the *Programming Handbook* for more information on PROTERMCAP and national language support.

UNBUFFERED

Reads one character at a time from a normally buffered data source, such as a file. You should use the UNBUFFERED option only when the input operations of a UNIX process invoked with the PROGRESS UNIX statement may be intermingled with the input being done by the PROGRESS statements that follow the INPUT THROUGH statement.

EXAMPLES

```
r-ithru.p

DEFINE VARIABLE process-id AS CHARACTER.
DEFINE VARIABLE dir-path AS CHARACTER  FORMAT "x(100)".

INPUT THROUGH echo $$ $PATH NO-ECHO.

SET process-id dir-path WITH FRAME indata
  NO-BOX NO-LABELS WIDTH 100.
DISPLAY process-id dir-path FORMAT "x(70)".

INPUT CLOSE.
```

This procedure uses as its input source the output of the UNIX echo command. Before the command is run, the UNIX shell substitutes the process-id number in place of $$ and the current directory search path for $PATH. The results are then echoed and become available as a line of input to PROGRESS. When the SET statement is executed, the line of input from echo is read and the values are assigned to the two variables. Those variables can then be used for any purpose. In this example, the word "echo" must be lowercase and the word "$PATH" must be uppercase as they are both passed to UNIX.

Whenever you use INPUT THROUGH, the UNIX program you name is executed as a separate process under its own shell. Therefore, the values of shell variables (such as $$) are values from that shell rather than the shell in which PROGRESS is executing.

r-ithru2.p

```
    DEFINE VARIABLE fn AS CHARACTER FORMAT "x(14)".

➤   INPUT THROUGH ls NO-ECHO.

    REPEAT:
      SET fn WITH NO-BOX NO-LABELS FRAME indata.
      DISPLAY fn.
    END.

    INPUT CLOSE.
```

This procedure gets input from the UNIX ls command. The ls command supplies the name of each UNIX file in your current directory. After the variable fn has been set, it is displayed.

NOTES

- Since BTOS/CTOS, DOS, and VMS environments do not have an implementation of pipes, the PROGRESS command INPUT THROUGH, which is included to interact with piping, is not operational. However, if you use this statement in a PROGRESS procedure, the procedure will compile. In addition, the procedure will run as long as the flow of control does not pass through the unsupported statement.

- INPUT THROUGH specifies the source for subsequent statements that process input. It does not actually read any data from the source.

- Data is read into the PROGRESS procedure using the INSERT, PROMPT-FOR, SET, or UPDATE statements. The data is placed in the frame fields referenced in these statements, and if ECHO is in effect then the frame is output to the current output destination. If you use the NO-ECHO option, then the frame is not output. If a subsequent DISPLAY statement causes the frame to be displayed, then the input data is also displayed.
- When INPUT THROUGH is closed, the pipe to the UNIX process is also closed.

SEE ALSO DEFINE STREAM Statement, INPUT FROM Statement, Chapter 9 of the *Programming Handbook*

INPUT-OUTPUT CLOSE Statement

Closes a specified or default stream opened by an INPUT-OUTPUT THROUGH statement.

SYNTAX

```
INPUT-OUTPUT [ STREAMstream ] CLOSE
```

STREAM *stream*

The name of the stream you want to close. If you do not name a stream, PROGRESS closes the default stream being used by an INPUT-OUTPUT THROUGH statement.

EXAMPLE

```
/* r-iothru.p */

FOR EACH item WHERE item-num < 10:
  DISPLAY item-num cost LABEL "Cost before recalculation".
END.

INPUT-OUTPUT THROUGH r-iothru UNBUFFERED.

FOR EACH item WHERE item-num < 10:
  EXPORT cost.
  SET cost.
END.

INPUT-OUTPUT CLOSE.
FOR EACH item WHERE item-num < 10  WITH COLUMN 40:
  DISPLAY item-num cost LABEL "Cost after recalculation".
END.
```

This procedure uses a C program to recalculate the cost of each item in inventory. Specifically, the C program increases the cost of each item by 3% or by 50 cents, whichever is greater. The INPUT-OUTPUT THROUGH statement tells the procedure to get its input from, and send its output to, the r-iothru program. The INPUT-OUTPUT CLOSE statement resets the input source to the terminal and the output destination to the terminal.

NOTES

- Since BTOS/CTOS, DOS, and VMS environments do not have an implementation of pipes, the PROGRESS command INPUT-OUTPUT CLOSE, which is included to interact with piping, is not operational. However, if you use this statement in a PROGRESS procedure, the procedure will compile. In addition, the procedure will run as long as the flow of control does not pass through the unsupported statement.

SEE ALSO DEFINE STREAM Statement, INPUT-OUTPUT-THROUGH Statement, Chapter 9 of the *Programming Handbook*

INPUT-OUTPUT THROUGH Statement

Names a UNIX program (process) for PROGRESS to start. This process becomes the input source as well as the output destination for the procedure.

SYNTAX

```
INPUT-OUTPUT [ STREAM stream ] THROUGH

{ program-name        } [ argument            ] ...  [ ECHO                  ] ...
{ VALUE( expression ) } [ VALUE( expression ) ]      [ NO-ECHO               ]
                                                     [ UNBUFFERED            ]
                                                     [ MAP protermcap-entry  ]
                                                     [ [NO-] MAP             ]
```

STREAM *stream*

Specifies the name of a stream. If you do not name a stream, the unnamed stream is used. See the DEFINE STREAM reference page and Chapter 9 of the *Programming Handbook* for more information about streams.

program-name

The name of the UNIX program from which the procedure is getting data and to which the procedure is sending data.

VALUE*(expression)*

An expression (a constant, field name, variable name, or any combination of these) whose value is the name of a UNIX program from which the procedure is getting data and to which the procedure is sending data.

argument

An argument you want to pass to the UNIX program. INPUT-OUTPUT THROUGH passes this argument as a character string.

If the argument is the literal value "echo", "no-echo", or "unbuffered" you must enclose it in quotes to prevent PROGRESS from interpreting that argument as one of the ECHO, NO-ECHO, or UNBUFFERED options for the INPUT-OUTPUT THROUGH statement.

VALUE*(expression)*

An expression (a constant, field name, variable name, or any combination of these) whose value is an argument you want to pass to the UNIX program. INPUT-OUTPUT THROUGH passes the value of *expression* as a character string.

ECHO

Displays all input data to the unnamed stream. Data is not echoed by default.

NO-ECHO

Accepts input data without displaying it on the current unnamed stream. Data is not echoed by default.

[NO-] MAP *protermcap-entry*

The *protermcap-entry* is an entry from the PROTERMCAP file. MAP enables you to send output to and receive input from an I/O stream that uses different character translation than the current stream. Typically, *protermcap-entry* is a slash-separated combination of a standard device entry and one or more language-specific add-on entries (MAP `laserwriter/french` or MAP `hp2/spanish/italian`, for example). PROGRESS uses the protermcap entries to build a translation table for the stream. Use NO-MAP to make PROGRESS bypass character translation altogether. See also Chapter 2 in the *Programming Handbook* for more information on PROTERMCAP and national language support.

UNBUFFERED

Reads and writes one character at a time to and from a normally buffered data source, such as a file. Use the UNBUFFERED option only when the input-output operations of a process invoked with PROGRESS' UNIX statement may be intermingled with the input-output being done by the PROGRESS statements that follow the INPUT-OUTPUT THROUGH statement. That is, INPUT-OUTPUT THROUGH handles the buffering of data between the PROGRESS procedure and the UNIX program that it invokes. You should use the UNBUFFERED option if your procedure invokes any other programs with the UNIX statement.

EXAMPLE

r-iothru.p

```
FOR EACH item WHERE item-num < 10:
  DISPLAY item-num cost LABEL "Cost before recalculation".
END.

INPUT-OUTPUT THROUGH r-iothru UNBUFFERED.

FOR EACH item WHERE item-num < 10:
  EXPORT cost.
  SET cost.
END.

INPUT-OUTPUT CLOSE.
FOR EACH item WHERE item-num < 10  WITH COLUMN 40:
  DISPLAY item-num cost LABEL "Cost after recalculation".
END.
```

This procedure uses a C program to recalculate the cost of each item in inventory. Specifically, the C program increases the cost of each item by 3% or by 50 cents, whichever is greater. The INPUT-OUTPUT THROUGH statement tells the procedure to get its input from, and send its output to, the r-iothru program. The INPUT-OUTPUT CLOSE statement resets the input source to the terminal and the output destination to the terminal.

You could certainly do this calculation within a single PROGRESS procedure, but the C program is used here only for illustration purposes. You would typically use a UNIX program outside PROGRESS to do specialized calculations or processing.

NOTES

- You must unpack the C program from the proguide subdirectory and compile it before you can use it with the r-iothru.p procedure. If you do not have a C compiler, you cannot try this example.

Here is the C program used by the r-iothru.p procedure:

r-iothru.c

```
#include <stdio.h>
#define MAX(a,b)    ( (a < b) ? b : a )

main()
{
        float  cost;

        /* This is important so that buffering does not   */
        /* defeat attempts to actually write on the pipe. */
        setbuf(stdout, (char *) NULL);

        while (scanf("%f", &cost) == 1) {
            /* Here the item cost is manipulated.  We are */
            /* increasing the cost by a fixed percentage  */
            /* (with a minimum increase), to provide      */
            /* an example.                                */
            cost = cost + MAX( 0.03 * cost, .50);
            printf("%10.2f\n", cost);
        }
}
```

NOTES

- Use EXPORT or PUT, not DISPLAY, to write data to the program.

- Since BTOS/CTOS, DOS, and VMS environments do not have an implementation of pipes, the PROGRESS command INPUT-OUTPUT THROUGH, which is included to interact with piping, is not operational. However, if you use this statement in a PROGRESS procedure, the procedure will compile. In addition, the procedure will run as long as the flow of control does not pass through the unsupported statement.

- Use SET to read data in from the program.

- If it is a C program, you may want to put an upper limit on how many errors can occur before the program ends. Also remember that if the program prints an error message, that message is sent to PROGRESS as data. You can use "fprintf(stderr,...)" to display debugging messages to the screen even in the middle of an INPUT-OUTPUT THROUGH operation.

- With INPUT-OUTPUT THROUGH in "non-interactive" mode, a PROGRESS procedure can send some information to a UNIX program, the program can process that information and send the results back to PROGRESS. Possible UNIX utilities you might use in batch mode are wc (word count) and sort.

  Here are some pointers for using INPUT-OUTPUT THROUGH in this way:

  - When the procedure finishes sending data to the program, use the OUTPUT CLOSE statement to reset the standard output stream to the screen. Doing this signals an EOF on the pipe, indicating that the program has received all input. When the procedure has finished receiving all data from the program, use the INPUT CLOSE statement to reset the standard input stream. Do not use the INPUT-OUTPUT CLOSE statement because that will close both pipes at once.

  - If you want to use the INPUT-OUTPUT THROUGH statement with a UNIX utility that buffers its output, you should use the non-interactive approach.

  - To signal an EOF, use OUTPUT CLOSE (rather than attempting to send a CTRL-D).

  Using INPUT-OUTPUT THROUGH in "interactive" mode involves PROGRESS sending some data to the program, the program sending a piece of data back to PROGRESS, and so on.

Here are some pointers for using INPUT-OUTPUT THROUGH in this way:

- At the end of the interaction between the procedure and the program, use the INPUT-OUTPUT CLOSE statement to shut down both pipes.
- Be sure that the program you are using does not buffer its output. If the program is a C program, the first line of the program should be "setbuf(stdout, (char *) NULL):". The program should also include "#include < stdio.h > ". These tell UNIX that the standard output of the program is unbuffered. If the program does buffer its output, you should use the batch approach to INPUT-OUTPUT THROUGH as explained in the above note.
- If the program ends on some condition other than upon detecting an EOF, be sure that it tells the PROGRESS procedure that it is about to end.

SEE ALSO DEFINE STREAM Statement, INPUT CLOSE Statement, INPUT-OUTPUT-CLOSE Statement, OUTPUT CLOSE Statement, Chapter 9 of the *Programming Handbook*

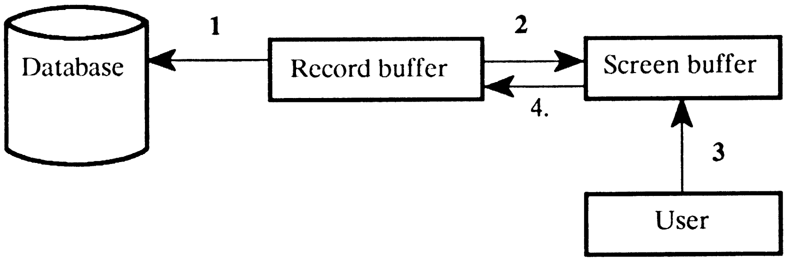
INSERT Statement

Creates a new database record, displays the initial values for the fields in the record, prompts for values of those fields, and assigns those values to the record.

The INSERT statement is a combination of the following statements:

CREATE	Creates an empty record buffer.
DISPLAY	Moves the record from the record buffer into the screen buffer (displaying the contents of the buffer on the screen).
PROMPT-FOR	Accepts input from the user, putting that input into the screen buffer.
ASSIGN	Moves data from the screen buffer into the record buffer.

DATA MOVEMENT



SYNTAX

```
INSERT record [EXCEPT field ...] [ frame-phrase ] [USING RECID ( n ) ]
```

record

The name of the record you want to add to a database file. PROGRESS creates one "record buffer" for every file you use in a procedure. This buffer is used to hold a single record from the file associated with the buffer. (You can use the DEFINE BUFFER statement to create additional buffers if necessary.) The CREATE part of the INSERT statement creates an empty record buffer for the file into which you are inserting a record.

To insert a record in a file defined for multiple databases, you must qualify the record's filename with the database name. See the description of the Record Phrase for more information.

frame-phrase

Specifies the overall layout and processing properties of a frame.

Here is the syntax of *frame-phrase*:

```
      [ ACCUM                                                  ]
      [ ATTR-SPACE                                             ]
      [ CENTERED                                               ]
      [       { [ DISPLAY ] color-phrase }                     ]
      [ COLOR { PROMPT color-phrase      } ...                 ]
      [ COLUMN expression                                      ]
      [ n COLUMNS                                              ]
      [ DOWN                                                   ]
      [ expression    DOWN                                     ]
      [ FRAME frame                                            ]
WITH  [ NO-ATTR-SPACE                                          ] ...
      [ NO-BOX                                                 ]
      [ NO-HIDE                                                ]
      [ NO-LABELS                                              ]
      [ NO-UNDERLINE                                           ]
      [ NO-VALIDATE                                            ]
      [ OVERLAY                                                ]
      [ PAGE-BOTTOM                                            ]
      [ PAGE-TOP                                               ]
      [ RETAIN n                                               ]
      [ ROW expression                                         ]
      [ SCROLL n                                               ]
      [ SIDE-LABELS                                            ]
      [ TITLE [ COLOR   color-phrase     ] expression          ]
      [ TOP-ONLY                                               ]
      [ WIDTH n                                                ]
```

For more information on *frame-phrase*, see the Frame Phrase reference page.

EXCEPT *field*

All fields except those fields listed in the EXCEPT phrase will be inserted.

USING RECID (*n*)

Enables you to insert a record into an RMS relative file using a specific record number, where *n* is the record-id of the record you want to insert.

EXAMPLE

```
                                                        r-insrt.p

   REPEAT:
 ➛   INSERT order WITH 1 COLUMN.
     REPEAT:
       CREATE order-line.
       order-line.order-num = order.order-num.
       UPDATE line-num order-line.item-num qty price.
       /* Verify the item-num by finding an item
          with that number */
       FIND item OF order-line.
     END.
   END.
```

Here, the user adds a new order record. After the user adds a new order record, the procedure creates order-lines for that record. The procedure uses the CREATE statement to create order-lines rather than the INSERT statement. When you use the INSERT statement, the PROMPT-FOR and ASSIGN parts of the INSERT let you put data into all the fields of the record being inserted. In the case of order-lines, this procedure only lets you add information into a few of the order-line fields. By using CREATE together with UPDATE, you can single out just those fields.

NOTES

- If an error occurs during the INSERT statement (e.g. the user enters a duplicate index value for a unique index), PROGRESS retries the data entry part of the statement and does not do the error processing associated with the block containing the statement.

- Any frame characteristics described by an INSERT statement contribute to the frame definition. When PROGRESS compiles a procedure, it uses a top to bottom pass of the procedure to design all the frames needed by that procedure, including those referenced by any INSERT statements, adding field and related format attributes as it goes through the procedure.

- If you are getting input from a device other than the terminal, and the number of characters read by the INSERT statement for a particular field or variable exceeds the display format for that field or variable, PROGRESS returns an error. However, if you are setting a logical field that has a format of "y/n" and the data file contains a value of "yes" or "no", PROGRESS converts that value to "y" or "n".

- If you use a single qualified identifier with the INSERT statement, the compiler first interprets the reference as *dbname.filename*. If the compiler cannot resolve the reference as *dbname.filename*, it tries to resolve it as *filename.fieldname*.

  When inserting fields, you must use filenames that are different from field names to avoid ambiguous references. See the description of the Record Phrase for more information.

SEE ALSO DEFINE BUFFER Statement, Frame Phrase

INSERT INTO Statement (SQL)

Adds new rows to a table.

SYNTAX

INSERT INTO *table-name* [(*column-list*)]
{{VALUES (*value-list*)} | *SELECT-statement* }

INTO *table-name*
The name of the table into which you want to insert new rows.

[*column-list*]
The names of one or more columns into which you want to insert values. If you specify a column list that omits one or more columns in the table, the omitted columns will have null values. If you omit the column list, PROGRESS/SQL assumes you want to insert the data into all columns in the target table in left-to-right order.

VALUES *value-list*
Specifies the values to be inserted into the table. Each value can be a literal, a defined PROGRESS variable or field name, or the keyword NULL. Each character string literal must be enclosed in either single or double quotation marks. The *n*th value in the VALUES clause is inserted into the *n*th column position in the INTO clause.

SELECT-statement
Selects values from columns in another table for insertion into the target table.

EXAMPLE

```
INSERT INTO employee
    (emp_num, name, job, dept, hiredate, salary)
    VALUES ('1204', 'Robins, Lynn', 'Mgr.',
        'Research', NULL, NULL).
```

NOTE

- The INSERT INTO statement can be used in both interactive SQL and embedded SQL.

INTEGER Function

Converts an expression of any data type to an integer value, rounding that value if necessary.

SYNTAX

```
INTEGER( expression )
```

expression

A constant, field name, variable name or any combination of these whose value can be of any data type. If the value of *expression* is character then it must be valid for conversion into a number (e.g. "1.67" is valid, "1.x3" is not). If *expression* is logical then the result is 0 if *expression* is false and 1 if *expression* is true. If *expression* is a date, then the result is the number of days from 1/1/4713 B.C. to that day. If the value of *expression* is the unknown value (?), then the result is also unknown.

EXAMPLE

r-intgr.p

```
    DEFINE VARIABLE new-loc AS INTEGER
      LABEL "new loc".

    FOR EACH item:
 ➤    new-loc = INTEGER(SUBSTRING(loc,5)).
      DISPLAY item-num loc new-loc.
    END.
```

This procedure converts a character field in your database to an integer field. All the items in the item file have a location code (loc) that contains the word "Bin" followed by a number. The procedure determines a new location code for each item by removing the word "Bin" from the front of the code. The SUBSTRING function extracts the location code, starting with the fifth character. The INTEGER function converts the remaining code to an integer value.

SEE ALSO DECIMAL Function, STRING Function

IS-ATTR-SPACE Function

Returns a logical value that indicates if the current terminal type is spacetaking or non-spacetaking. Returns a value of YES if the terminal is spacetaking, or a value of NO if the terminal is non-spacetaking.

SYNTAX

```
IS-ATTR-SPACE
```

EXAMPLE

```
                                                   r-isattr.p

    DEFINE VARIABLE termtype AS LOGICAL
      FORMAT "spacetaking/non-spacetaking".

    TERMINAL = "wy60".
 ➤  termtype = IS-ATTR-SPACE.
    DISPLAY "You are currently using a" termtype NO-LABEL
     "terminal" WITH FRAME d1 CENTERED ROW 5.
```

In this procedure, the terminal type is set to "wy60". Since "wy60" is a non-spacetaking terminal, the IS-ATTR-SPACE function evaluates to FALSE, and the procedure variable `termtype` equals "non-spacetaking".

NOTES

- If you are running PROGRESS in batch mode, IS-ATTR-SPACE returns a value of ?.

SEE ALSO TERMINAL Statement, Chapter 7 of the *Programming Handbook*

KBLABEL Function

Returns the keyboard label (such as F1) of the standard key that performs a specified PROGRESS function (such as GO).

SYNTAX

```
KBLABEL( key-function )
```

key-function

An expression (a constant, field name, variable name, or any combination of these) whose value is the name of the special PROGRESS key function. See Chapter 2 of the *Programming Handbook* for possible values of *key-name.* If *key-function* is a constant, you must enclose it in quotation marks (""). See the same chapter for a list of key functions and the corresponding standard keyboard keys.

EXAMPLE

r-kblabl.p

```
➤ STATUS INPUT "Enter data, then press "  + KBLABEL("GO").
  FOR EACH customer:
    UPDATE cust-num name address city st.
  END.
```

The r-kblabl.p procedure lets the user update some of the fields in each of the customer records. It also displays a message in the "status" message area at the bottom of the screen. This message reads "Enter data, then press" followed by the keyboard label of the key that performs the GO function.

KEYCODE Function

Evaluates a key label (such as F1) for a key in the predefined set of keyboard keys and returns the corresponding integer key code (such as 301). See Chapter 2 of the *Programming Handbook* for a list of key codes and key labels.

SYNTAX

```
KEYCODE( key-label )
```

key-label

An expression (a constant, field name, variable name or any combination of these) whose character value is the label of the key for which you want the key code. If *key-label* is a constant, you must enclose it in quotation marks ("").

EXAMPLE

r-keycod.p

```
DEFINE VARIABLE msg AS CHARACTER EXTENT 3.
DEFINE VARIABLE i AS INTEGER INITIAL 1.
DEFINE VARIABLE newi AS INTEGER INITIAL 1.

DISPLAY "    Please choose     " SKIP(1)
        " 1 Run order entry    " @ msg[1]
          ATTR-SPACE SKIP
        " 2 Run receivables    " @ msg[2]
          ATTR-SPACE SKIP
        " 3 Exit               " @ msg[3]
          ATTR-SPACE SKIP
        WITH CENTERED FRAME menu NO-LABELS.

REPEAT:
  COLOR DISPLAY MESSAGES msg[i] WITH FRAME menu.
  READKEY.
  IF LASTKEY = KEYCODE("CURSOR-DOWN") AND i < 3
  THEN newi = i + 1.
  ELSE IF LASTKEY = KEYCODE("CURSOR-UP") AND i > 1
  THEN newi = i - 1.
  ELSE IF LASTKEY = KEYCODE("F1") OR
     LASTKEY = KEYCODE("RETURN")
  THEN LEAVE.

  IF i < > newi THEN COLOR DISPLAY NORMAL
    msg[i] WITH FRAME menu.
  i = newi.
END.
```

This procedure displays a menu and highlights different selections on the menu depending on which key you press. On the first iteration of the REPEAT block, the COLOR statement tells PROGRESS to color msg[i] with the same color used to display messages. Because the initial value of i is 1, msg[i] is the first menu selection. Therefore, the first menu selection is colored MESSAGES. Here's what happens if you press the cursor-down key:

1. The READKEY statement reads the value of the key you pressed.
2. The first IF statement tests to see if the keycode of the key you pressed is "CURSOR-DOWN". It also checks to see if the value of i is less than 3. Both of these things are true, so the procedure adds 1 to the value of newi, making newi equal 2.
3. The next two IF statements are ignored because the condition in the first IF statement was true. The procedure continues on to the last IF statement: IF i < > newi THEN COLOR DISPLAY NORMAL msg[i] WITH FRAME menu.
4. Remember, i is still 1 but newi is now 2. So i is not equal to newi. That means that the IF statement test is true. Therefore, PROGRESS colors msg[i], which is still msg[1] (the first menu selection), NORMAL. So the first menu selection is no longer highlighted.
5. Just before the end of the REPEAT block, i is set equal to newi. That means that msg[i] is now msg[2] or the second menu selection.
6. On the next iteration, the COLOR statement colors msg[i], which is the second menu selection, MESSAGES. You can see that the end result of pressing the cursor-down key is that the highlight bar is moved to the second menu selection.

SEE ALSO KEYCODE Statement, Chapter 2 of the *Programming Handbook*

KEYFUNCTION Function

Evaluates an integer expression (such as 301) and returns a character string that is the function of the key associated with that integer expression (such as GO). See Chapter 2 parameter of the *Programming Handbook* for a list of key functions.

SYNTAX

```
KEYFUNCTION( expression )
```

expression
: A constant, field name, variable name, or any combination of these whose value is an integer key-code.

EXAMPLE

r-keyfn.p

```
DEFINE VARIABLE msg AS CHARACTER EXTENT 3.
DEFINE VARIABLE i AS INTEGER INITIAL 1.
DEFINE VARIABLE newi AS INTEGER.
DEFINE VARIABLE func AS CHARACTER.

DISPLAY "    Please choose     " SKIP(1)
        " 1 Run order entry    " @ msg[1]
          ATTR-SPACE SKIP
        " 2 Run receivables    " @ msg[2]
          ATTR-SPACE SKIP
        " 3 Exit               " @ msg[3]
          ATTR-SPACE SKIP
        WITH CENTERED FRAME menu NO-LABELS.

REPEAT:
  COLOR DISPLAY MESSAGES msg[i] WITH FRAME menu.
  READKEY.
➔ func = KEYFUNCTION(LASTKEY).
  IF func = "CURSOR-DOWN" AND i < 3
  THEN newi = i + 1.
  ELSE IF func = "CURSOR-UP" AND i > 1
  THEN newi= i - 1.
  ELSE IF func = "GO" OR func = "RETURN"
  THEN LEAVE.
  IF i <> newi THEN COLOR DISPLAY NORMAL
    msg[i] WITH FRAME menu.
  i = newi.
END.
```

This procedure displays a menu and highlights different selections depending on which key you press. On the first iteration of the REPEAT block, the COLOR statement tells PROGRESS to color msg[i] with the same color used to display messages. Because the initial value of i is 1, msg[i] is the first menu selection. Therefore, the first menu selection is colored MESSAGES.

Suppose you press the cursor-down key. Here's what happens:

1. The READKEY statement reads the value of the key you pressed.

2. The first IF statement tests to see if the function of the key you pressed is "CURSOR-DOWN". It also checks to see if the value of i is less than 3. Both of these things are true, so the procedure adds 1 to the value of newi, making newi equal 2.

3. The next two IF statements are ignored because the condition in the first IF statement was true. The procedure continues on to the last IF statement: IF i < > newi THEN COLOR DISPLAY NORMAL msg[i] WITH FRAME menu.

4. Remember, i is still 1 but newi is now 2. So i is not equal to newi. That means that the IF statement test is true. Therefore, PROGRESS colors msg[i], which is still msg[1] (the first menu selection), NORMAL. So the first menu selection is no longer highlighted.

5. Just before the end of the REPEAT block, i is set equal to newi. That means that msg[i] is now msg[2] or the second menu selection.

6. On the next iteration, the COLOR statement colors msg[i], which is the second menu selection, MESSAGES. You can see that the end result of pressing the cursor-down key is that the highlight bar is moved to the second menu selection.

NOTES

- The value returned by the KEYFUNCTION function is affected by any ON statements you may have used to redefine the value of the key represented by *expression*.

- If the key represented by *expression* has no function currently assigned to it or if it has the function of BELL, KEYFUNCTION returns a null value.

- KEYFUNCTION(-2) is equal to ENDKEY.

SEE ALSO Chapter 2 of the *Programming Handbook*

KEYLABEL Function

Evaluates a key-code (such as 301) and returns a character string that is the predefined keyboard label for that key (such as F1).

SYNTAX

```
KEYLABEL( key-code )
```

key-code

The key code of the key whose label you want to know. A special case of *key-code* is LASTKEY. See Chapter 2 of the *Programming Handbook* for a list of key codes and key labels.

EXAMPLE

r-keylbl.p

```
DISPLAY "Press the F1 key to leave procedure".
REPEAT:
  READKEY.
  HIDE MESSAGE.
  IF LASTKEY = KEYCODE("F1") THEN RETURN.
➔ MESSAGE "Sorry, you pressed the" KEYLABEL(LASTKEY) "key".
END.
```

This procedure reads each keystroke the user makes, leaving the procedure only when the user presses F1. The KEYLABEL function tests the LASTKEY pressed (remember that the value in LASTKEY is the key code of the last key pressed), and returns the label of the key.

NOTE

- Some key-codes may be associated with more than one key label. The KEYLABEL function always returns the label listed first in the PROGRESS table of key labels.

SEE ALSO KEYCODE Function, Chapter 2 of the *Programming Handbook*

KEYWORD Function

Returns a character value indicating whether a string is a PROGRESS keyword.

SYNTAX

```
KEYWORD(expression)
```

expression

A constant, field name, variable name, or any combination of these that results in a character string. If expression matches a PROGRESS keyword or valid abbreviation of a keyword, the KEYWORD function returns the full keyword. If there is no match, the KEYWORD function returns the value "?".

If expression is either "def" (the abbreviation for DEFINE) or "col" (the abbreviation for COLUMN), the KEYWORD function returns the values "def" and "col" respectively.

If you are using PROGRESS Run-Time, the KEYWORD function always returns the value "?".

EXAMPLE

r-keywd.p

```
DEFINE formname AS CHARACTER FORMAT "x(20)".

REPEAT ON ERROR UNDO, RETRY:
  UPDATE formname.
➤ IF KEYWORD(formname) NE ?
  THEN DO:
    MESSAGE formname + " may not be used as a form name".
    UNDO, RETRY.
  END.
  ELSE LEAVE.
END.
```

NOTES

- Because KEYWORD recognizes abbreviations, it does not distinguish between "FORM" and "FORMAT" or between "ACCUM" and "ACCUMULATE."
- This function returns the unknown value for colors and any one or two character abbreviation for a data type.

LAST Function

Rcturns a TRUE value if the current iteration of a DO, FOR EACH, or REPEAT...BREAK block is the last iteration of that block.

SYNTAX

```
LAST( break-group )
```

break-group

The name of a field or expression you named in the block header with the BREAK BY option.

EXAMPLE

r-last.p

```
FOR EACH item BY on-hand * cost DESCENDING:
  DISPLAY item-num on-hand * cost (TOTAL) LABEL "Value-oh".
END.

FOR EACH item BREAK BY on-hand * cost
  DESCENDING:
    FORM item.item-num value-oh AS DECIMAL
          LABEL "Value-oh" WITH COLUMN 40.
    DISPLAY item-num on-hand * cost @ value-oh.
    ACCUMULATE on-hand * cost (TOTAL).
➜  IF LAST(on-hand * cost) THEN DO:
      UNDERLINE value-oh.
      DISPLAY ACCUM TOTAL on-hand * cost  @ value-oh.
    END.
END.
```

The first FOR EACH block produces a list of the on-hand values of the items in inventory. It also automatically generates a total of these on-hand values.

The second FOR EACH block does exactly the same thing, except the total is not generated by PROGRESS. Instead, the procedure uses the ACCUMULATE statement and the LAST function. By doing it this way, you can substitute your own labels and formatting for the grand total.

SEE ALSO FIRST Function, FIRST-OF Function, LAST-OF Function

LASTKEY Function

Returns the integer key code of the most recent key pressed during an interaction with a procedure.

SYNTAX

```
LASTKEY
```

EXAMPLE

r-lastky.p

```
DISPLAY "You may update each customer."
        "After making your changes," SKIP
        "Press one of:" SKIP(1)
        " F1 - Make the changes permanent"  SKIP
        " F4 - Undo changes and exit"       SKIP
        " F9 - Undo changes and try again"  SKIP
        " F10- Find next customer"          SKIP
        " F11- Find previous customer"
        WITH CENTERED FRAME instr.

FIND FIRST customer.
REPEAT:
  UPDATE cust-num name address city st
         GO-ON(F9 F10 F11) WITH 1 DOWN.
  IF LASTKEY = KEYCODE("F9")
  THEN UNDO, RETRY.
  ELSE
  IF LASTKEY = KEYCODE("F10")
  THEN FIND NEXT customer.
  ELSE
  IF LASTKEY = KEYCODE("F11")
  THEN FIND PREV customer.
END.
```

In this procedure, the user can move through the customer file and update certain fields in each of the customer records. The GO-ON function tells the procedure to continue on to the next statements if the user presses F9, F10, or F11. To determine what action to take, the LASTKEY function compares the keycode of the last key pressed with the keycodes F9, F10, and F11.

NOTES

- If you used a READKEY statement that timed out (you specified a number of seconds by using the PAUSE option with the READKEY statement), or if a PAUSE statement times out, the value of LASTKEY is -1.
- If you use the PAUSE option with the READKEY statement, the value of LASTKEY is the key you press to end the PAUSE.
- When PROGRESS starts, the value of LASTKEY is -1. This value remains the same until the first input, READKEY, or procedure pause occurs. LASTKEY is reset to -1 each time you return to the PROGRESS editor.
- If you read data from a file, LASTKEY is set to the last character read from the file. For an INSERT, PROMPT-FOR, SET or UPDATE statement, this is always KEYCODE("RETURN"). For a READKEY statement, this is the character read from the file. If you reach past the end of the file, LASTKEY will be -2.

SEE ALSO READKEY Statement, Chapter 2 of the *Programming Handbook*

LAST-OF Function

Returns a TRUE value if the current iteration of a DO, FOR EACH, or REPEAT...BREAK block is the last iteration for a particular value of a break group.

SYNTAX

```
LAST-OF( break-group )
```

break-group

The name of a field or expression you named in the block header with the BREAK BY option.

EXAMPLE

r-lastof.p

```
FOR EACH item BREAK BY prod-line:
  ACCUMULATE on-hand * cost (TOTAL BY prod-line).
➤ IF LAST-OF(prod-line)
  THEN DISPLAY prod-line (ACCUM TOTAL BY prod-line
                 on-hand * cost) LABEL "Value-oh".
END.
```

This procedure uses LAST-OF to display a single line of information about each prod-line group in the item file, without displaying any individual item data. It produces a report showing the aggregate value on-hand for each product line.

SEE ALSO FIRST Function, FIRST-OF Function, LAST Function

LC Function

Returns a character string identical to a specified string, but with all uppercase letters in that string converted to lowercase.

SYNTAX

```
LC ( string )
```

string

A character expression (a constant, field name, variable name or any combination of these whose value is a character), containing uppercase letters you want to convert to lowercase.

EXAMPLE

r-lc.p

```
REPEAT:
   PROMPT-FOR customer.cust-num.
   FIND customer USING cust-num.
   DISPLAY name.
   UPDATE sales-region.
   sales-region =
      CAPS (SUBSTRING (sales-region, 1, 1)) +
 ➤    LC (SUBSTRING (sales-region, 2)).
   DISPLAY sales-region.
END.
```

This procedure finds a customer record. After the user updates the sales-region field, the procedure converts the first character of the sales-region value to uppercase and the remaining characters to lowercase.

The CAPS function uses the SUBSTRING function to extract the first character of the field, which it then converts to uppercase.

In the LC function, the result of the SUBSTRING function is the remaining characters in the sales-region filed, starting with character position 2. (No length is specified, so the remainder of the string is assumed). The LC function converts these characters to lowercase.

SEE ALSO CAPS function

LDBNAME Function

The LDBNAME function returns the logical name of a currently connected database.

SYNTAX

$$\text{LDBNAME}\left\{\begin{array}{l}(\textit{integer-expression})\\(\textit{logical-name})\\(\textit{alias})\end{array}\right\}$$

integer-expression

If the parameter supplied to LDBNAME is an integer expression, and there are, for example, three currently connected databases, then LDBNAME(1), LDBNAME(2), and LDBNAME(3) return their logical names. Also, continuing the same example of three connected databases, LDBNAME(4), LDBNAME(5), etc., return the ? value.

logical-name or *alias*

These forms of the LDBNAME function require a quoted character string or a character expression as a parameter. If the parameter is the logical name of a connected database or an alias of a connected database then the logical name is returned. Otherwise, the ? value is returned.

EXAMPLE

r-ldbnm.p

```
DO WHILE LDBNAME(1) <> ? : /* the parm. is the number 1 */
   DISCONNECT VALUE(LDBNAME(1)).
END.
```

This procedure disconnects all currently connected databases. After a database is disconnected, the connected databases are re-numbered to reflect the change. For example, if databases 1, 2, 3, 4, are connected and then database 3 is disconnected, database 4 becomes database 3.

NOTE

- If you want to know if a particular name is an ALIAS or a true logical database name, then use the following procedure:

r-tstnm.p

```
DEF VAR testnm as char.
SET testnm.
IF LDBNAME (testnm) = testnm
   THEN MESSAGE testnm "is a true logical database name.".
   ELSE
      IF LDBNAME (testnm) = ?
         THEN MESSAGE
         testnm "is not the name or alias of any connected database.".
         ELSE MESSAGE
         testnm "is an ALIAS for database " LDBNAME(testnm).
```

SEE ALSO CONNECT, DISCONNECT, CREATE ALIAS, and DELETE ALIAS statements; CONNECTED, PDBNAME, DBTYPE, DBRESTRICTIONS, DBMIS, FRAME-DB, and NUM-DBS functions.

LE or < = Operator

Returns a TRUE value if the first of two expressions is less than or equal to the second.

SYNTAX

$$\textit{expression} \begin{Bmatrix} \text{LE} \\ <= \end{Bmatrix} \textit{expression}$$

expression

A constant, field name, variable name, or any combination of theses, The expressions on either side of the LE or < = must be of the same data type, although one may be integer and the other decimal.

EXAMPLE

r-le.p

```
FOR EACH item WHERE on-hand <= 0:
      DISPLAY item.item-num idesc on-hand.
END.
```

This procedure lists all the items with zero or negative on-hand quantities.

NOTE

- When comparing two character expressions, the LE function treats lowercase letters as if they were uppercase letters.

LEAVE Statement

Exits from a block. Execution continues with the first statement after the end of the block.

SYNTAX

```
LEAVE [ label ]
```

label

The name of the block you want to leave. If you do not name a block, PROGRESS leaves the innermost iterating block that contains the LEAVE statement. If there is no such block, then PROGRESS leaves the procedure block.

r-leave.p

```
DEFINE VARIABLE valid-choice AS CHARACTER INITIAL "NPFQ".
DEFINE VARIABLE selection AS CHARACTER FORMAT "x".
main-loop:
REPEAT:
   choose:
   REPEAT ON ENDKEY UNDO choose, RETURN:
      MESSAGE "(N)ext (P)rev (F)ind (Q)uit"
         UPDATE selection AUTO-RETURN.
      IF INDEX(valid-choice, selection) <> 0
      THEN LEAVE choose. /*Selection was valid */
      BELL.
   END.  /* choose */

/* Processing for menu choices N, P, F here */

   IF selection = "Q" THEN LEAVE main-loop.
END.
```

This procedure represents part of a menu program. If the user chooses N, P, F, or Q, the procedure leaves the inner “choose” block and goes on to process the menu selection. If the user presses any other key, the procedure rings the bell.

SEE ALSO NEXT statement

LENGTH Function

Returns the length of a character string or the number of bytes in a raw datatype expression.

SYNTAX

```
LENGTH{(string)(raw expression)}
```

string

A character expression (a constant, field name, variable name, or any combination of these that results in a character string value).

raw expression

A function or variable name that returns a raw value.

EXAMPLE

r-length.p

```
DEFINE VARIABLE short-desc AS CHARACTER
  FORMAT "x(11)" LABEL "Desc".

FOR EACH item:
  IF LENGTH(idesc) > 8 THEN
    short-desc = SUBSTRING(idesc,1,8) + "...".
  ELSE short-desc = idesc + "...".
  DISPLAY item-num short-desc loc on-hand alloc
          rop oorder cost FORMAT "$>>9.99"
          subs-item.
END.
```

This procedure produces a report containing item information. Because the information on the report fills the entire width of the screen, this procedure shortens the information in the description field for each item. If the description of an item is longer than 8 characters, the description is converted to just the first 8 characters followed by ellipses.

rawlen.p

```
/*You must connect to a non-PROGRESS demo database to run
  this procedure*/

DEFINE VAR i AS INT.

FIND cust WHERE cust-num = 29.
i = LENGTH(RAW(name)).
DISPLAY i.
```

In this procedure, the LENGTH function returns the number of bytes in the name of number 29. The procedure returns a 13, the number of bytes in the name Chip's Poker.

NOTES

- LENGTH of the unknown value is the unknown value.

LENGTH Statement

Changes the number of bytes in a raw variable.

SYNTAX

```
LENGTH(variable) = expression
```

variable
A variable of the datatype raw.

expression
A constant, field name, variable name, or any combination of these that returns an integer.

EXAMPLE

rawlen1.p

```
/*You must connect to a non-PROGRESS demo database to run
  this procedure*/

DEFINE VAR r1 as RAW.

FIND cust WHERE cust-num = 29.
r1 = RAW(name).
LENGTH(r1) = 2.
```

This procedure takes the number of bytes in the name stored in the variable r1 and truncates it to two bytes.

NOTES

- If *variable* is the unknown value, it remains unknown.
- If *integer expression* is greater than the number of bytes in *variable*, PROGRESS appends nulls so that the length of *variable* equals the length of *integer expression*.

LIBRARY Function

The LIBRARY function parses a character string in the form *path-name<<member-name>>*, where *path-name* is the pathname of a library and *member-name* is the name of a file within the library, and returns the name of the library. The brackets << >> indicate that *member-name* is a file in a library. If the string is not in this form, the LIBRARY function returns an unknown value (?).

Generally, you will want to use the LIBRARY function with the SEARCH function to retrieve the name of a library. The SEARCH function returns character strings of the form *path-name<<member-name>>* if it finds a file in a library.

SYNTAX

```
LIBRARY ( string )
```

string
: A character expression (a constant, field name, variable or any combination of these that results in a character value) whose value is the pathname of a file in a library.

EXAMPLE

```
DEFINE VARIABLE what-lib AS CHARACTER.
DEFINE VARIABLE location AS CHARACTER.

location = SEARCH ("myfile.r").
IF location = ? THEN DO:
    MESSAGE "Can't find myfile.r".
    LEAVE.
END.

what-lib = LIBRARY(location).
IF what-lib <> ? THEN
    MESSAGE "myfile.r can be found in library" what-lib.
ELSE
    MESSAGE "myfile.r is not in a library but is in" location.
END.
```

This procedure searches for the file `myfile.r`. If it can't find the file, it displays the message "`Can't find myfile.r`." However, if it does find the file, it passes the pathname of the file to the LIBRARY function. If the file is in a library, the procedure displays the name of the library, and if the file is not in a library, the procedure displays the pathname of the file returned by SEARCH.

SEE ALSO MEMBER Function, SEARCH Function

LINE-COUNTER Function

Returns the current line number of paged output. Returns 0 if the output is not paged.

SYNTAX

```
LINE-COUNTER[ ( stream ) ]
```

stream

Specifies the name of a stream. If you do not name a stream, PROGRESS uses the unnamed stream. See the DEFINE STREAM reference page and Chapter 9 of the *Programming Handbook* for more information about streams.

EXAMPLE

r-linec.p

```
IF LOOKUP(KEYFUNCTION(LASTKEY),
OUTPUT TO PRINTER.
FOR EACH customer BREAK BY st:
  DISPLAY cust-num name address city st.
  IF LAST-OF(st) THEN DO:
    IF LINE-COUNTER + 4 > PAGE-SIZE
    THEN PAGE.
    ELSE DOWN 1.
  END.
END.
```

This procedure prints a customer report, categorized by state. At the end of each state category, it tests to see whether there are at least four lines left on the page. The LINE-COUNTER function returns the current line number of output. If that number plus 4 is greater than the total number of lines on the page (returned by the PAGE-SIZE function), then the procedure starts the new page. If there are four or more lines left, the procedure skips a line before printing the next customer record.

NOTES

- When output is going to a device other than the terminal screen, PROGRESS defers displaying a frame until a display is done to another frame. That way, if you do several consecutive displays to the same frame, PROGRESS performs all those displays at once. Because of this optimization, if the last display (a frame actually output) filled the page, the value returned by the LINE-COUNTER function can be larger than the page-size even though the next frame is printed at the start of the new page.

- To check that output is positioned on the first non-header line of a new page, use the following kind of procedure:

```
DEFINE VARIABLE newpage AS LOGICAL INITIAL YES.
DEFINE STREAM output1.

FOR EACH customer:
  FORM HEADER "Page Header" PAGE-NUMBER(output1)
              "Line" LINE-COUNTER(output1)
              WITH FRAME one PAGE-TOP NO-LABELS NO-BOX.
  VIEW STREAM output1 FRAME one.
  DISPLAY STREAM output1 name PAGE-NUMBER(output1)
    LINE-NUMBER(output1) WITH NO-LABELS NO-BOX.
  IF new-page THEN DISPLAY STREAM output1 "First Line".
  IF LINE-COUNTER(output1) > PAGE-SIZE(output1)
  THEN newpage = YES.
  ELSE newpage = NO.
END.
```

SEE ALSO DEFINE STREAM Statement, Chapter 9 of the *Programming Handbook*

LOCKED Function

Returns a TRUE value if a record is not available to a prior FIND...NO-WAIT statement because another user has locked a record,

SYNTAX

```
LOCKED record
```

record

The name of a record or buffer.

To use the LOCKED function with a record in a file defined for multiple databases, you must qualify the record's filename with the database name. See the description of the Record Phrase for more information.

EXAMPLE

r-locked.p

```
    REPEAT:
      PROMPT-FOR customer.cust-num.
      FIND customer USING customer.cust-num
        NO-ERROR NO-WAIT.
      IF NOT AVAILABLE customer THEN DO:
➡       IF LOCKED customer
        THEN MESSAGE "Customer record is locked".
        ELSE MESSAGE "Customer record not found".
        NEXT.
      END.
      DISPLAY cust-num name city st.
    END.
```

The FIND statement in this procedure tries to retrieve a customer record according to a supplied customer number. Because of the NO-ERROR option, the FIND statement does not return an error if it cannot find the record. The NO-WAIT option causes FIND to return immediately if the record is in use by another user.

The two main reasons a record might not be available are that it is locked (being used by another user) or does not exist. The LOCKED function returns a TRUE value if the record is locked. In this case, the r-locked.p procedure displays a message saying that the record is locked. If the record is not locked, the procedure displays a message saying that the record does not exist.

SEE ALSO AMBIGUOUS Function, AVAILABLE Function, FIND Statement, NEW Function

LOG Function

Calculates the logarithm of an expression using a specified base.

SYNTAX

```
LOG( expression[ ,base ] )
```

expression

A decimal expression (a constant, field name, variable name or any combination of these whose value is decimal) for which you want the logarithm.

base

A numeric expression that is the base you want to use. If you do not specify a base, LOG returns the natural logarithm, base (e). *base* must be greater than 1.

EXAMPLE

```
r-log.p

DEFINE VARIABLE base AS DECIMAL
  FORMAT ">>>,>>>.9999".
DEFINE VARIABLE number AS DECIMAL.
REPEAT:
  UPDATE base
    VALIDATE(base > 1,"Base must be greater than 1").
  REPEAT:
    UPDATE number
      VALIDATE(number > 0,"Number must be positive").
➤   DISPLAY number LOG(number, base)
      LABEL "LOG(NUMBER, BASE)".
  END.
END.
```

This procedure prompts the user for a base and a number and then displays the log of the number. The VALIDATE options on the UPDATE statements ensure that the user enters a base value greater than 1 and a number greater than 0.

NOTES

- The LOG function provides precision of about 15 decimal places.
- After converting the base and exponent to floating-point format, the LOG function uses standard system routines. On certain machines, the logarithm routines do not handle very large numbers well and may cause your terminal to hang.

LOOKUP Function

Returns an integer giving the position of an expression in a list. Returns a 0 if the expression is not in the list.

SYNTAX

```
LOOKUP ( expression,list )
```

expression

A constant, field name, variable name, or any combination of these that results in a character value that you want to look up within a list of character expressions. The comparison between *expression* and *list* is case-insensitive. If the value of *expression* is unknown (?), the result of the LOOKUP function is unknown.

list

A list of character expressions (constants, field names, variable names, or any combination of these that result in a character value) that contains the expression you name with the *expression* argument. Separate each entry in *list* with a comma. If *list* is the unknown value (?), the result of LOOKUP is unknown.

EXAMPLE

r-lookup.p

```
DEFINE VARIABLE stlist AS CHARACTER
  INITIAL "ME,MA,VT,RI,CT,NH".
DEFINE VARIABLE state AS CHARACTER FORMAT "x(2)".

REPEAT:
  SET state LABEL
    "Enter a New England state, 2 characters".
  IF LOOKUP(state,stlist) = 0
  THEN MESSAGE "This is not a New England state".
END.
```

This procedure prompts the user for one of the New England states. The LOOKUP function tests the value against the list of states stored in the stlist variable. If there is no match (the result is 0), the procedure displays a message. Otherwise, the procedure prompts the user for another New England state.

NOTE

- If *expression* contains commas, LOOKUP returns the beginning of a series of entries in *list*. For example, `LOOKUP("a,b,c","x,a,b,c")` returns a 2.
- Most character comparisons are case-insensitive in PROGRESS. By default, all characters are converted to upper case prior to comparisons. However, it is possible to define fields and variables as case sensitive (though it is not advised, unless strict ANSI SQL adherence is required). If the *expression* is a field or variable defined to be case-sensitive, the *list* is also treated as case-sensitive, and "Smith" does not match "smith."

SEE ALSO ENTRY Function, INDEX Function

LT or < Operator

Returns a TRUE value if the first of two expressions is less than the second.

SYNTAX

$$\textit{expression} \begin{Bmatrix} \text{LT} \\ < \end{Bmatrix} \textit{expression}$$

expression

A constant, field name, variable name, or any combination of these. The expressions on either side of the LT or < = must be of the same data type, although one can be integer and the other can be decimal.

EXAMPLE

```
r-lt.p

FOR EACH item WHERE on-hand < alloc:
  DISPLAY item.item-num idesc on-hand alloc.
END.
```

This procedure displays information for those item records whose on-hand value is less than the allocated value.

NOTES

- You can compare character strings with LT. Most character comparisons are case-insensitive in PROGRESS. That is, all characters are converted to upper case prior to comparisons. However, it is possible to define fields and variables as case-sensitive (though it is not advised, unless strict ANSI SQL adherence is required). If either *expression* is a field or variable defined to be case-sensitive, the comparison is case-sensitive and "Smith" does not equal "smith."

  Note that because characters are converted to their ASCII values for comparison, all upper case letters sort before all lower case letters (**a** is greater than **Z**, but less than **b**).

MATCHES Function

Compares a character expression to a pattern and returns a TRUE value if the expression satisfies the pattern criteria.

SYNTAX

```
expression MATCHES pattern
```

expression

A character expression (a constant, field name, variable name, or any combination of these whose value is character) that you want to check to see if it conforms with the *pattern*.

pattern

A character expression that you want to match with the *string*. This may include a constant, field name, variable name, or any combination of these whose value is a character.

The *pattern* can contain wildcard characters: a period (.) in a particular position indicates that any single character is acceptable in that position; an asterisk (*) indicates that any group of characters is acceptable, including a null group of characters.

EXAMPLE

```
                                                    r-match.p
FOR EACH customer WHERE address MATCHES("*St"):
  DISPLAY name address city st zip.
END.
```

This procedure displays customer information for all customers whose address ends in St. An index is not used for the customer search in r-match.p.

NOTES

- When comparing two character values, MATCHES treats lowercase letters as if they were uppercase letters.

- Most character comparisons are case-insensitive in PROGRESS. By default, all characters are converted to upper case prior to comparisons. However, it is possible to define fields and variables as case sensitive (though it is not advised, unless strict ANSI SQL adherence is required). If the *expression* preceding the MATCHES keyword is a field or variable defined to be case-sensitive, the comparison is case-sensitive. In a case-sensitive comparison "SMITH" does not match "Smith."

- If in the pattern you want to specify a period or asterisk as a literal character rather than a wildcard character, enter a tilde (~) before the character. For example, the result of "*a.b" MATCHES "~*a.~.b" is TRUE. If you specify the match pattern as a literal quoted string in a procedure file, enter each tilde as two tildes so they will be interpreted as tildes for the match pattern.

SEE ALSO BEGINS Function

MAXIMUM Function

Compares two values and returns the larger of the two values.

SYNTAX

```
MAXIMUM( expression1,expression2 )
```

expression1
A constant, field name, variable name, or any combination of these whose value is of the same data type as *expression2*.

expression2
A constant, field name, variable name, or any combination of these whose value is of the same data type as *expression1*.

EXAMPLE

r-maxmum.p

```
DEFINE VARIABLE max-cred2 AS DECIMAL
  FORMAT ">>,>>9.99".

FOR EACH customer:
  IF max-credit < 2000
  THEN max-cred2 = max-credit + 1000.
  ELSE max-cred2 = 5000.
  DISPLAY max-credit max-cred2
          MAXIMUM(max-cred2,max-credit)
          LABEL "Maximum of these two values".
END.
```

In this procedure, if the max-credit value is under 2000, the procedure adds 1000 to that value. Otherwise, the procedure sets max-credit to 5000. The MAXIMUM function determines the greater of the original max-credit value and the new max-cred2 value.

NOTE

- When comparing two character values, MAXIMUM treats lowercase letters as if they were uppercase letters.

SEE ALSO MINIMUM Function

MEMBER Function

The MEMBER function parses a character string in the form *path-name<<member-name>>*, where *path-name* is the pathname of a library and *member-name* is the name of a file within the library, and returns *member-name*. The brackets << >> indicate that *member-name* is a file in a library. If the string is not in this form, the MEMBER function returns an unknown value (?).

Generally, you will want to use the MEMBER function with the SEARCH function to determine if a file is in a library. If a data file is in a library, you must first extract the file from the library in order to read it (see "Building Libraries with the Prolib Utiltiy" in chapter 4 of *System Administration II: General* for more information about extracting a file from a library). The SEARCH function returns a character string of the form *path-name<<member-name>>* if it finds a file in a library.

SYNTAX

```
MEMBER ( string )
```

string

A character expression (a constant, field name, variable or any combination of these that results in a character value) whose value is the pathname of a file in a library.

EXAMPLE

```
DEFINE INPUT PARAMETER filename AS CHARACTER.
DEFINE VARIABLE what-lib AS CHARACTER.
DEFINE VARIABLE location AS CHARACTER.

location = SEARCH(filename).

IF location = ? THEN DO:
    MESSAGE "Can't find file" filename.
    LEAVE.
END.

IF LIBRARY(location) <> ? THEN
    MESSAGE "File"
        MEMBER(location) "is in library" LIBRARY(location).
    LEAVE.
ELSE
    MESSAGE "File is not in a library but is in" location.
END.
```

This procedure is passed an input parameter whose value is the name of a file. Using this value, the procedure searches for the file and if it can't find the file, it displays a message and ceases operation. If it does find the file, it tests to see if the file is in a library. If so, the procedure displays the filename and the name of the library. Otherwise, the procedure displays the pathname of the file returned by SEARCH.

SEE ALSO LIBRARY Function, SEARCH Function

MESSAGE Statement

Displays messages in the message area at the bottom of the screen. The bottom line of the screen is reserved for PROGRESS system messages. The two lines above that are reserved for messages you display with the MESSAGE statement. The MESSAGE statement displays messages on the first of these two lines unless that line is already occupied, in which case MESSAGE uses the second of the two lines.

SYNTAX

```
MESSAGE [ COLOR color-phrase ] [ expression ] ...

    [ { SET    } field [ FORMAT string  ] ]
    [ { UPDATE }       [ AUTO-RETURN    ] ]
```

COLOR *color-phrase*

Displays a message using the color you specify with the Color phrase. Here is the syntax for *color-phrase*:

```
{ NORMAL                                                                }
{ INPUT                                                                 }
{ MESSAGES                                                              }
{ protermcap-attribute                                                  }
{ dos-hex-attribute                                                     }
{ [ BLINK- ][BRIGHT-][ fgnd-color ] [ / bgnd-color ]                    }
{ [ BLINK- ][RVV- ][ UNDERLINE- ][BRIGHT- ] [fgnd-color ]               }
{ VALUE(expression)                                                     }
```

For more information about *color-phrase*, see the Color phrase reference page.

expression

An expression (a constant, field name, variable name, or any combination of these) whose value you want to display in the message area. If *expression* is not character, it is converted to character before it is displayed. If you do not use this option, you must use either the SET or UPDATE option.

SET *field*

Displays the *expression* you specified and SETs the field or variable you name (prompts the user for input and assigns the value entered to the field or variable). You cannot test the field with either the ENTERED function or the NOT ENTERED function.

UPDATE *field*

Displays the *expression* you specified and updates the field or variable you name (displays the current value of the field or variable, prompts for input, and assigns the value entered to the field or variable). You cannot test the field with either the ENTERED function or the NOT ENTERED function.

FORMAT *string*

The format in which you want to display the *expression*. For more information on display formats, see Chapter 4 of the *Programming Handbook*. If you do not use the FORMAT option, PROGRESS uses the defaults shown in Table 20.

Table 20: Default Display Formats

Type of Expression	**Default Format**
Field	Format from Dictionary
Variable	Format from variable definition
Constant character	Length of character string
Other	Default format for the data type of the expression. Table 21 shows these default formats.

Table 21: Default Data Type Display Formats

Data Type of Expression	**Default Format**
Character	x(8)
Date	99/99/99
Decimal	->>,>>9.99
Integer	->,>>>,>>9
Logical	yes/no

AUTO-RETURN

Performs a carriage return when the field being SET or UPDATEd is full.

EXAMPLE

r-msg.p

```
REPEAT:
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num NO-ERROR.
  IF NOT AVAILABLE customer
  THEN DO:
    MESSAGE "Customer with cust-num "
      INPUT cust-num
      " does not exist.  Please try another.".
    UNDO, RETRY.
  END.
  ELSE DO:
    DISPLAY name sales-region.
  END.
END.
```

In this procedure, if you enter the number of a customer that does not exist, the procedure displays a message telling you the customer does not exist. If the customer does exist, the procedure displays the name and sales-region of the customer.

NOTES

- The MESSAGE statement always sends messages to the current output destination. If the INPUT source is the terminal, messages are displayed on the screen and are also sent to the current output destination. Compiler error messages follow this convention also.

- If you don't want messages sent to the current output destination, redirect the output to a named stream. PROGRESS never writes messages to a named stream.

  If you are sending output to a destination other than the terminal, and you don't want messages to appear on the terminal (and if you are not really using the terminal as an input source), use one of the following statements:

Operating System	INPUT FROM
UNIX	`INPUT FROM /dev/null`
DOS & OS/2	`INPUT FROM NUL.`
VMS	`INPUT FROM NL:`
BTOS/CTOS	`INPUT FROM [NUL]`

Be sure to use the INPUT CLOSE statement to close the input source.

- PROGRESS automatically clears messages after any user interaction, such as a SET, UPDATE, or PAUSE statement, but not after a READKEY statement.

- When you use the MESSAGE SET or MESSAGE UPDATE statement to update a field, PROGRESS does not process any validation criteria defined for that field in the database. For example, if the validation criteria for the customer.name field is:

  ```
  Valexp: name BEGINS "a"
  ```

  and you use the statement

  ```
  MESSAGE UPDATE name
  ```

  PROGRESS lets you enter any data, including data that does not start with an "a", into the name field.

  Use the MESSAGE statement to display a message, but use the SET or UPDATE statement to let the user change the data in a frame rather than in the message area.

- If the combination of the text and field you name in a MESSAGE UPDATE statement exceeds 80 characters, PROGRESS truncates the text to fit on the message line. For example:

  ```
  DEFINE VARIABLE myvar AS CHARACTER FORMAT "x(60)".
  MESSAGE "abcdefghijklmnopqrstuvwxyz" UPDATE myvar.
  ```

  Here, the combination of the message text and the myvar variable exceeds 80 characters, so PROGRESS truncates the message text.

- Use of the MESSAGE statement to display decimal values results in the truncation of the non-significant zeros to the right of the decimal point. For example, this procedure

```
DEFINE VARIABLE amt AS DECIMAL FORMAT ">>9.99"
INITIAL 1.20.
MESSAGE "Total" amt.
```

displays the message:

```
"Total 1.2"
```

Use functions such as STRING and DECIMAL to control the format of a display.

SEE ALSO Color Phrase, DECIMAL function, Format Phrase, INTEGER function, STRING Function, Chapter 4 of the *Programming Handbook*

MESSAGE-LINES Function

Returns the number of lines in the message area at the bottom of the terminal screen (always 2).

SYNTAX

```
MESSAGE-LINES
```

MINIMUM Function

Compares two values and returns the smaller of the two.

SYNTAX

```
MINIMUM ( expression1,expression2 )
```

expression1
A constant, field name, variable name, or any combination of these whose value is of the same data type as *expression2.*

expression2
A constant, field name, variable name, or any combination of these whose value is of the same data type as *expression1.*

EXAMPLE

r-minmum.p

```
DEFINE VARIABLE want LIKE on-hand
  LABEL "How many do you want?".
DEFINE VARIABLE ans AS LOGICAL.

REPEAT:
  PROMPT-FOR item.item-num want.
  FIND item USING item-num.
  ans = no.
  IF MINIMUM(INPUT want,on-hand) = INPUT want
  THEN DO:
    MESSAGE "We have enough" idesc "in stock.".
    MESSAGE
      "Any other items to check?" UPDATE ans.
    IF NOT ans THEN LEAVE.
  END.
  ELSE DO:
    MESSAGE "We only have" on-hand idesc
            "in stock.".
    MESSAGE "Any other items to check?"
            UPDATE ans.
    IF NOT ans THEN LEAVE.
  END.
END.
```

This procedure prompts users for an item number and how many of the item they want. If the number of items a user wants (stored in the want variable) is the minimum of the want variable and the on-hand field, the procedure displays a message saying there is enough in stock. Otherwise, the procedure displays a message saying there is not enough in stock.

NOTE

- When comparing two character values, MINIMUM treats lowercase letters as if they were uppercase letters.

SEE ALSO MAXIMUM Function

MODULO Function

Returns the remainder after division.

SYNTAX

```
expression MODULO base
```

expression

An integer expression (a constant, field name, variable name, or any combination of these whose value is integer).

base

A positive integer expression (a constant, field name, variable name, or any combination of these whose value is integer) that is the modulo base. For example, angles measured in degrees use a base of 360 for modulo arithmetic. 372 MODULO 360 is 12.

EXAMPLE

r-modulo.p

```
REPEAT:
  SET qty-avail AS INTEGER LABEL "Qty.Avail.".
  SET std-cap AS INTEGER
              LABEL "Std. Truck Capacity".
  DISPLAY TRUNCATE(qty-avail / std-cap,0)
    FORMAT ">,>>9" LABEL "# Full Loads"
    qty-avail MODULO std-cap LABEL "Qty.Left.".
END.
```

This procedure determines the number of trucks required to ship a given quantity of material, and how much is left over for a less than full truck load.

NOTE

- *expression* must be greater than 0 in order for MODULO to return a correct value.

MONTH Function

Returns the month value (from 1 to 12) of a date you specify.

SYNTAX

```
MONTH( date )
```

date

A date expression (a constant, field name, variable name or any combination of these whose value is a date) for which you want a month value.

EXAMPLE

```
r-mon.p

FOR EACH order:
  IF (MONTH(pdate) <= MONTH(TODAY) OR
     YEAR(pdate) < YEAR(TODAY))
    AND sdate = ?
  THEN DISPLAY order-num LABEL "Order Num"
               cust-po LABEL "P.O. Num"
               pdate LABEL "Promised By"
               odate LABEL "Ordered"
               terms
       WITH TITLE "These orders are overdue".
END.
```

This procedure displays all the orders that have a promise date in a month that has passed, and whose sdate (ship-date) field is unknown (the initial value of the sdate field).

SEE ALSO DAY Function, WEEKDAY Function, YEAR Function

NE or < > Operator

Compares two expressions and returns a TRUE value if they are not equal.

SYNTAX

```
expression  { NE  }  expression
            { < > }
```

expression

A constant, field name, variable name, or any combination of these. The expressions on either side of the NE or < > must be of the same data type.

EXAMPLE

```
r-ne.p

➤ FOR EACH item WHERE subs-item < > 0:
    DISPLAY item-num idesc subs-item
            LABEL "Alternate item".
  END.
```

This procedure displays information for all items that have a substitute item (the subs-item field is not equal to 0).

NOTES

- If one of the expressions has an unknown value and the other does not, the result is TRUE. If both are unknown, the result is FALSE. The exception to this is SQL. For SQL, if either or both expressions is unknown, then the result is unknown.

- You can compare character strings with NE. Most character comparisons are case-insensitive in PROGRESS. That is, all characters are converted to upper case prior to comparisons. However, it is possible to define fields and variables as case-sensitive (though it is not advised, unless strict ANSI SQL adherence is required). If either *expression* is a field or variable defined to be case-sensitive, the comparison is case-sensitive and "Smith" does not equal "smith."

NEW Function

Checks a record buffer and returns a TRUE value if the record in that buffer was newly created. If the record was read from the database, NEW returns a FALSE value.

SYNTAX

```
NEW record
```

record

The name of the record buffer you want to check with the NEW function.

To use the NEW function with a record in a file defined for multiple databases, you must qualify the record's filename with the database name. See the description of the Record Phrase for more information.

EXAMPLE

r-new.p

```
REPEAT:
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num NO-ERROR.
  IF NOT AVAILABLE customer
  THEN DO WITH FRAME newcus:
    MESSAGE "Creating new customer".
    CREATE customer.
    ASSIGN cust-num.
    UPDATE name address city st zip.
  END.

  CREATE order.
  order.cust-num = customer.cust-num.
  IF NEW customer THEN DO:
    UPDATE order.order-num pdate.
    order.terms = "COD".
    DISPLAY order.terms.
  END.
  ELSE UPDATE order.order-num pdate order.terms.
END.
```

This partial procedure enters new orders, optionally creating a customer record if one does not exist. The NEW function is later used to select alternate processing depending on whether a customer is newly created or already exists.

NOTE

- The NEW function returns a TRUE value only during the transaction in which the record is created. If the scope of the record is greater than the transaction in which that record is created, the NEW function returns a FALSE value outside the transaction.

SEE ALSO AVAILABLE Function, FIND Statement, LOCKED Function

NEXT Statement

Goes directly to the END of an iterating block and starts the next iteration of the block.

SYNTAX

```
NEXT [ label ]
```

label

The name of the block for which you want to start the next iteration. If you do not name a block, PROGRESS starts the next iteration of the innermost iterating block that contains the NEXT statement.

EXAMPLE

r-next.p

```
PROMPT-FOR customer.sales-rep
  LABEL "Enter salesman initials"
  WITH SIDE-LABELS CENTERED.

FOR EACH customer:
  IF sales-rep <> INPUT sales-rep THEN NEXT.
  DISPLAY cust-num name city st
    WITH CENTERED.
END.
```

The FOR EACH block in this procedure reads a single customer record on each iteration of the block. If the sales-rep field of a customer record does not match the sales-rep value supplied to the PROMPT-FOR statement, the NEXT statement causes PROGRESS to do the next iteration of the FOR EACH block, bypassing the DISPLAY statement.

SEE ALSO LEAVE Statement

NEXT-PROMPT Statement

The NEXT-PROMPT statement specifies which field to position the cursor on during the next input operation involving that field in a frame.

SYNTAX

```
NEXT-PROMPT field [ frame-phrase ]
```

field

The name of the input field on which you want to place the cursor the next time the user supplies input to the frame. If the field you name is not an input field in the frame, PROGRESS disregards the NEXT-PROMPT statement.

frame-phrase

Specifies the overall layout and processing properties of a frame. Here is the syntax for *frame-phrase*:

```
        [ ATTR-SPACE                                          ]
        [ CENTERED                                            ]
        [         { [ DISPLAY ] color-phrase }                ]
        [ COLOR   { PROMPT color-phrase      } ...            ]
        [ COLUMN expression                                   ]
        [ n COLUMNS                                           ]
        [ DOWN                                                ]
        [ expression DOWN                                     ]
        [ FRAME frame                                         ]
        [ NO-ATTR-SPACE                                       ]
        [ NO-BOX                                              ]
WITH    [ NO-HIDE                                             ] ...
        [ NO-LABELS                                           ]
        [ NO-UNDERLINE                                        ]
        [ NO-VALIDATE                                         ]
        [ OVERLAY                                             ]
        [ PAGE-BOTTOM                                         ]
        [ PAGE-TOP                                            ]
        [ RETAIN n                                            ]
        [ ROW expression                                      ]
        [ SCROLL n                                            ]
        [ SIDE-LABELS                                         ]
        [ TITLE [ COLOR color-phrase ] expression             ]
        [ TOP-ONLY                                            ]
        [ WIDTH n                                             ]
```

For more information on *frame-phrase*, see the Frame Phrase reference page.

EXAMPLE

```
r-nprmpt.p

FOR EACH customer:
  UPDATE customer WITH 2 COLUMNS.
  IF tax-no EQ " "
  THEN DO:
    MESSAGE "You must enter a tax number".
➤   NEXT-PROMPT tax-no.
    UNDO, RETRY.
  END.
END.
```

This procedure lets you update customer information. If you do not enter a value for tax-no, then PROGRESS positions the cursor at the tax-no field when the UPDATE statement is processed following the UNDO, RETRY of the FOR EACH block.

NOTES

- NEXT-PROMPT is especially useful in an EDITING phrase because it can dynamically reposition the cursor depending on input from the user.
- When you need to do complex field checking that you are unable to do either in a Dictionary validation expression or in a VALIDATE Frame phrase, you can use NEXT-PROMPT to position the cursor after detecting an error.
- If the next data entry statement involving the frame specified with NEXT-PROMPT does not use the indicated NEXT-PROMPT field, then PROGRESS ignores the NEXT-PROMPT statement.
- The NEXT-PROMPT statement can affect default frame layout. For instance, in the following procedure PROGRESS prompts for "a" and "b" (in that order).

```
r-nextp.p

DEF VAR a AS CHAR.
DEF VAR b AS CHAR.
UPDATE a b.
```

However, if you include NEXT-PROMPT b before the update statement, as shown in the following procedure, PROGRESS prompts for "b" first and "a" second.

r-nextp1.p

```
DEF VAR a AS CHAR.
DEF VAR b AS CHAR.
NEXT-PROMPT b.
UPDATE a b.
```

SEE ALSO EDITING Phrase, Frame Phrase

NOT Function

Returns TRUE if an expression is false and FALSE if an expression is true.

SYNTAX

```
NOT expression
```

expression

A logical expression (a constant, field name, variable name or any combination of these whose value is logical (true/false, yes/no)).

EXAMPLE

r-not.p

```
REPEAT:
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num NO-ERROR.
  IF NOT AVAILABLE customer
  THEN DO:
    MESSAGE "Customer with cust-num:"
      INPUT cust-num
      " does not exist.  Please try another.".
    UNDO, RETRY.
  END.
  ELSE DO:
    DISPLAY name phone.
  END.
END.
```

In this procedure, if the user enters the number of a customer that does not exist, the procedure displays a message telling the user that the customer does not exist and to try again. If the customer does exist, the procedure displays the name and phone number of the customer.

SEE ALSO AND Operator, OR Function

NOT ENTERED Function

Returns a TRUE value if a frame field was not modified during the last INSERT, PROMPT-FOR, SET, or UPDATE statement which used the field.

SYNTAX

```
field  NOT ENTERED
```

field
The name of the field or variable you are checking.

EXAMPLE

r-nenter.p

```
DEFINE VARIABLE new-max LIKE max-credit.

FOR EACH customer:
    DISPLAY cust-num name max-credit
      LABEL "Current max credit"
      WITH FRAME a 1 DOWN ROW 1.
    SET new-max LABEL "New max credit"
      WITH SIDE-LABELS NO-BOX ROW 10 FRAME b.
→   IF new-max NOT ENTERED OR new-max = max-credit
    THEN DO:
      DISPLAY "No Change in Max-Credit"
        WITH FRAME d ROW 15.
      NEXT.
    END.
    DISPLAY "Changing Max Credit of" name SKIP
            "from" max-credit "to"
            new-max WITH FRAME c ROW 15 NO-LABELS.
    max-credit = new-max.
 END.
```

This procedure displays the number, name, and max-credit for each customer. For each customer, the procedure prompts the user for a new max-credit value. The NOT ENTERED function tests to see whether you enter a value. If you enter a value and it is different from the present value of max-credit, the procedure displays the old and new max-credit values. If you enter the same value or no value, the procedure displays a message telling the you that the max-credit has not been changed.

SEE ALSO ENTERED Function

NUM-ALIASES Function

The NUM-ALIASES function returns an integer value representing the number of aliases defined. The NUM-ALIASES function uses no arguments.

SYNTAX

```
NUM-ALIASES
```

EXAMPLE

r-numal.p

```
DEF VAR I AS INT.
DISPLAY NUM-ALIASES LABEL "Number of Defined Aliases:".
REPEAT I = 1 TO NUM-ALIASES.
    DISPLAY ALIAS(I) LABEL "Alias"
        LDBNAME(ALIAS(I)) LABEL "Logical Database".
END.
```

This procedure displays the number of defined aliases. It also displays the aliases and logical names of all connected databases.

SEE ALSO CONNECT, DISCONNECT, CREATE ALIAS, and DELETE ALIAS statements; ALIAS, CONNECTED, DBTYPE, DBRESTRICTIONS, FRAME-DB, LDBNAME, PDBNAME, SDBNAME, and GATEWAYS functions.

NUM-DBS Function

The NUM-DBS function takes no arguments. It returns the number of connected databases.

SYNTAX

```
NUM-DBS
```

EXAMPLE

r-numdbs.p

```
DEFINE VARIABLE i AS INTEGER.
REPEAT i = 1 TO NUM-DBS:
   DISPLAY LDBNAME(i) DBRESTRICTIONS(i) FORMAT "x(40)".
END.
```

This procedure uses NUM-DBS to display the logical name and database restrictions of all connected databases.

SEE ALSO CONNECT, DISCONNECT, CREATE ALIAS, and DELETE ALIAS statements; CONNECTED, LDBNAME, PDBNAME, DBTYPE, DBRESTRICTIONS, FRAME-DB, SDBNAME, and GATEWAYS functions.

NUM-ENTRIES Function

Returns the number of items in a comma-separated list of strings.

SYNTAX

```
NUM-ENTRIES ( string-expression )
```

string-expression

A comma-separated list of strings--an "entry list" (see the ENTRY function). NUM-ENTRIES returns the number of elements in the list. Strictly, NUM-ENTRIES returns the number of commas plus 1, and it returns 0 if *string-expression* equals "".

EXAMPLE

r-n-ent1.p

```
   DEFINE VARIABLE i AS INTEGER.
   DEFINE VARIABLE regions AS CHARACTER INITIAL "Northeast,
     Southeast,Midwest,Northwest,Southwest".
➡ REPEAT i=1 TO NUM-ENTRIES(regions):
     DISPLAY ENTRY(i,regions) FORMAT "x(12)".
   END.
```

This procedure uses NUM-ENTRIES and ENTRY to loop through a list of regions and display them, one per line. This example is trivial, since there are obviously five regions; the simple REPEAT statement, REPEAT i=1 TO 5, would work fine here. Now look at the following example, where PROPATH is a comma-separated list of unknown length:

r-n-ent2.p

```
   DEFINE VARIABLE i AS INTEGER.
➡ REPEAT i=1 TO NUM-ENTRIES(PROPATH):
     DISPLAY ENTRY(i,PROPATH) FORMAT "x(64)".
   END.
```

This procedure uses NUM-ENTRIES to loop through the PROPATH (a comma-separated list of directory paths) and print the directories, one per line.

SEE ALSO ENTRY function.

ON Statement

Indicates an action to be taken when you press a special key (such as F1) in response to a SET, PROMPT-FOR, UPDATE, or INSERT statement or when a procedure pauses. A procedure pauses either because PROGRESS encounters a PAUSE statement or because the screen is full. To use the ON statement, choose the key for which you want to define a specific action. If you are using UNIX or VMS, be sure that the key is defined in the protermcap file.

SYNTAX

```
ON  key-label key-function
```

key-label

The label of the key for which you want to define a specific action. See Chapter 2 of the *Programming Handbook* for a list of key labels.

If you are using UNIX or VMS, all of the special PROGRESS keys have definitions in the protermcap file that was supplied with PROGRESS. If the key for which you are defining an action is not already in protermcap, you must add a definition for that key. Keys that you can name that do not require a protermcap definition are any CTRL key, RETURN, BACKSPACE, TAB, and DEL.

Under DOS and OS/2, keys are predefined as described in Chapter 2 of the *Programming Handbook.*

key-function

The action you want PROGRESS to take when the user presses the key associated with *key-label. Key-function* can be one of the following:

```
BACKSPACE            ENDKEY                PICK-AREA
BACK-TAB             ERROR                 PICK-LABEL-AREA
BELL                 GO                    RECALL
BOTTOM-COLUMN        GOTOLINE              REPAINT
CANCEL-PICK          HELP                  REPORTS
CHOICES              HOME                  RETURN
CLEAR                INSERT-COLUMN         RIGHT-END
CURSOR-DOWN          INSERT-FIELD          SCROLL-LEFT
CURSOR-LEFT          INSERT-FIELD-DATA     SCROLL-RIGHT
CURSOR-RIGHT         INSERT-FIELD-LABEL    SETTINGS
CURSOR-UP            INSERT-MODE           STOP
DELETE-CHARACTER     LEFT-END              STOP-DISPLAY
DELETE-COLUMN        MAIN-MENU             TAB
DELETE-FIELD         MOVE                  TOP-COLUMN
END-ERROR            PICK
```

EXAMPLE

```
r-onstmt.p

ON F1 HELP.
ON F2 GO.
ON CTRL-X BELL.
ON F4 ENDKEY.
```

This procedure reverses the GO and HELP keys (F1 and F2). It redefines CTRL-X to ring the bell instead of taking the usual GO action. It defines F4 to take the ENDKEY action. (By default, F4 is assigned to END-ERROR, not ENDKEY.)

NOTES

- Any new action you define for a key remains in effect until you use another ON statement for that key or until you return to the PROGRESS editor.
- If an input request comes from a READKEY statement, PROGRESS does not take the action specified by the ON statement.
- The ON statement does not affect the way function keys work while you are in the PROGRESS editor. For example, if you use the ON statement to define a new function for the RECALL key, that key will still work the normal way in the editor, recalling the last procedure that was run from the editor.
- The ON statement does not affect the way function keys work while you are using the PROGRESS Dictionary unless you start the Dictionary from within an application that uses the ON statement to redefine function keys.
- If you want to disable a key, you can use the ON statement to tell PROGRESS to ring the bell when the user presses that key. For example, suppose you want to disable the CLEAR key. You can use this statement to redefine the action PROGRESS takes when you press CLEAR :

  ```
  ON F8 BELL.
  ```

  F8 is the label of the CLEAR key. The label may be different on your terminal.

- If you want to reassign a function to a different key, use the ON statement to assign that function to a new key and use another ON statement to ring the bell when you press the original key. For example, suppose you want to reassign the INSERT function to the key with the label F12:

```
ON F12 INSERT.
ON F3 BELL.
```

  Here, F3 is the key which originally did the INSERT function.

- In an X window environment, you can use the keys on a mouse within an ON statement in a PROGRESS application. Currently, PROGRESS supports the mouse key-label LEFT-MOUSE-UP. For example, suppose you want to display help information when you press the left mouse key in a field:

```
ON LEFT-MOUSE-UP HELP.
```

ON ENDKEY Phrase

Describes the processing you want to take when the ENDKEY condition occurs during a block. This condition usually occurs when you press END-ERROR (F4) during the first interaction of a block iteration, or any time you press ENDKEY.

If you are using a REPEAT or FOR EACH block, the default processing when the ENDKEY condition occurs is to undo all of the processing that was done in the current iteration of the block and to leave the block, continuing on to any remaining statements in the procedure.

SYNTAX

```
ON ENDKEYUNDO  [ label1 ] [ , LEAVE   [ label2 ] ]
                          [ , NEXT    [ label2 ] ]
                          [ , RETRY   [ label2 ] ]
                          [ , RETURN             ]
```

label1

The name of the block whose processing you want to undo. If you do not name a block with *label1*, ON ENDKEY UNDO undoes the processing of the block started by the block start statement using the ON ENDKEY phrase.

LEAVE *label2*

Indicates that, after undoing the processing of a block, PROGRESS should leave the block labelled *label2*. If you do not name a block, PROGRESS leaves the block containing the ON ENDKEY phrase. After leaving a block, PROGRESS continues on with any remaining processing in a procedure.

LEAVE is the default if you do not specify any of LEAVE, NEXT, RETRY or RETURN.

NEXT *label2*

Indicates that, after undoing the processing of a block, PROGRESS should do the next iteration of the block you name with the label2 option. If you do not name a block with the NEXT option, PROGRESS does the next iteration of the block labelled *label1*.

RETRY *label2*

Indicates that, after undoing the processing of a block, PROGRESS should repeat the same iteration of the block you name with the *label2* option. If you give *label2*, it must be the same as the value for *label1*. If you do not give *label2*, it is assumed to be the same as *label1*.

When a block is retried, any frames scoped to that block are not advanced or cleared. For more information, see Chapter 7 of the *Programming Handbook*.

RETURN

Returns to the calling procedure or, if there is no calling procedure, to the PROGRESS editor.

EXAMPLES

r-endky.p

```
FOR EACH customer ON ENDKEY UNDO, RETRY:
  DISPLAY cust-num name max-credit.
  SET max-credit VALIDATE
    (max-credit > 0, "non-zero max-credit").
END.
```

In the r-endky.p procedure, if the user presses END-ERROR (F4) or ENDKEY while changing the max-credit field, any changes made during the current iteration of the block are undone, and the same iteration is run again. If this procedure did not use the ON ENDKEY phrase and the user pressed END-ERROR (F4), the procedure would end because the default ENDKEY action is UNDO, LEAVE. After leaving the FOR EACH block the procedure ends because there are no more statements.

r-endky2.p

```
customer:
  FOR EACH customer:
    DISPLAY cust-num.
    UPDATE max-credit.
    FOR EACH order OF customer
        ON ENDKEY UNDO customer, LEAVE customer:
      UPDATE order-num sdate.
    END.
  END.
```

In the first iteration of the customer block, the r-endky2.p procedure displays the cust-num and updates max-credit for the first customer. Then the second FOR EACH block gets the orders for that customer, letting the user update the order-num and sdate fields for each order. If the user presses END-ERROR (F4) while updating this order information, the procedure undoes the updates to the order information for all the customer's orders and for the customer record and leaves the customer block, ending the procedure.

If the inner FOR EACH block did not include the ON ENDKEY phrase, only updates to the current order would be undone when the user pressed END-ERROR (F4).

NOTES

- If ENDKEY processing is necessary within a DO block that does not have an ON ENDKEY phrase, then the ENDKEY processing specified (or that takes place by default) on the innermost FOR EACH, REPEAT, DO ON ENDKEY or procedure block that encloses the DO block is performed.

- The processing you describe with the ON ENDKEY phrase applies only to the block whose block statement (REPEAT, DO, or FOR EACH) contains the phrase. Any blocks within that block use their default END key processing or any processing you describe for them individually with the ON ENDKEY phrase.

- The processing you describe with the ON ENDKEY phrase applies only when the user presses the END-ERROR key during the first screen interaction of a block. At any other time when the user presses the END-ERROR key, PROGRESS takes the default ON ERROR action for the block or any specific action you describe with the ON ERROR phrase. ON ENDKEY always applies when the user presses $\boxed{\text{ENDKEY}}$.

- If nothing will be different during a RETRY of a block, then the RETRY is treated as a NEXT or a LEAVE. This default action provides protection against infinite loops.

SEE ALSO ON ERROR Phrase, UNDO Statement, Chapter 8 of the *Programming Handbook*

ON ERROR Phrase

Describes the processing you want to take place when there is an error during a block. If you are using a REPEAT block or a FOR EACH block, the default processing when there is an error during a block is to undo all of the processing that has been done in the current iteration of the block and to retry the block iteration during which the error occurred.

SYNTAX

```
                                 ┌, LEAVE  [ label2 ]┐
ON ERROR  UNDO[ label1 ]         │, NEXT   [ label2 ]│
                                 │, RETRY  [ label2 ]│
                                 └, RETURN           ┘
```

label1

The name of the block whose processing you want to undo. If you do not name a block with *label1*, ON ERROR UNDO undoes the processing of the block started by the block start statement using the ON ERROR phrase.

LEAVE *label2*

Indicates that after undoing the processing of a block, PROGRESS should leave the block labeled *label2*. If you do not name a block, PROGRESS leaves the block labeled with *label1*.

NEXT *label2*

Indicates that after undoing the processing of a block, PROGRESS should do the next iteration of the block you name with the label2 option. If you do not name a block with the NEXT option, PROGRESS does the next iteration of the block labelled with *label1*.

RETRY *label2*

Indicates that after undoing the processing of a block, PROGRESS should repeat the same iteration of the block you name with the *label2* option. If you give *label2*, it must be the same as the value for *label1*. If you do not give *label2*, it is assumed to be the same as *label1*.

RETRY is the default processing if you do not use any of LEAVE, NEXT, RETRY, or RETURN. When a block is being retried, any frames scoped to that block are not advanced or cleared. For more information, see Chapter 7 of the *Programming Handbook*.

RETURN

Returns to the calling procedure or, if there was no calling procedure, to the PROGRESS editor.

EXAMPLES

r-onerr.p

```
REPEAT ON ERROR UNDO, NEXT:
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num.
  DISPLAY name address city st zip.
END.
```

In r-onerr.p, if you enter a customer number and the FIND statement is unable to find a customer with that number, PROGRESS returns an error. If an error occurs, the ON-ERROR phrase tells PROGRESS to undo anything that was done in the current iteration and start the next iteration. This way, you see the numbers you entered that were invalid, but you can also go on to the next customer number you want to enter.

r-onerr2.p

```
ord:
  REPEAT:
    PROMPT-FOR order.order-num.
    FIND order USING order-num.
    DISPLAY odate sdate.
    FOR EACH order-line OF order
      ON ERROR UNDO ord, NEXT ord:
      FIND item OF order-line.
      qty-ship = qty.
      alloc = alloc - qty.
      on-hand = on-hand - qty.
      IF on-hand < 0 THEN UNDO ord, NEXT ord.
    END.
    MESSAGE "Order quantities processed".
  END.
```

The r-onerr2.p procedure asks you for an order number and displays information about that order. Then the procedure finds the item associated with each order-line on the order. If the FIND statement is unable to find a specific item record, PROGRESS returns an error. In this case, PROGRESS undoes all the work done in the outer REPEAT block and starts the next iteration of that block.

NOTES

- If error processing is necessary within a DO block that does not have an ON ERROR phrase on the DO statement, then the error processing specified, or that takes place by default, on the innermost FOR EACH, REPEAT, DO ON ERROR, DO TRANSACTION, or procedure block that encloses the DO block is performed.
- The processing you describe with the ON ERROR phrase applies only to the block whose block statement (REPEAT, DO, or FOR EACH) contains the phrase. Any blocks within that block use their default error processing or any processing you describe for them individually with the ON ERROR phrase.
- The default ON ERROR processing or any processing you describe with the ON ERROR phrase applies to any errors that may occur during a block. These errors include the case where a user presses the END-ERROR key any time after the first screen interaction of the block.
- If nothing will be different during a RETRY of a block, then the RETRY is treated as a NEXT or a LEAVE. This default action provides protection against infinite loops.

SEE ALSO ON ENDKEY Phrase, UNDO Statement, Chapter 8 of the *Programming Handbook*

OPEN Statement (SQL)

Selects the retrieval set from the execution of the SELECT clause in a DECLARE CURSOR statement.

SYNTAX

```
OPEN  cursor-name
```

cursor-name
The name given to the cursor where defined in the DECLARE cursor statement.

EXAMPLE

```
OPEN c01.
```

NOTE

- The OPEN statement can be used in both interactive SQL and embedded SQL.

OPSYS Function

Identifies the operating system being used, so that a single version of a procedure can work differently under different operating system. Returns a value of MSDOS if you are running PROGRESS on a DOS system; returns a value of OS2 if you are running PROGRESS on an OS/2 system; returns a value of UNIX if you are running PROGRESS on a UNIX or XENIX system; returns a value of VMS if you are running PROGRESS on a VMS system; returns a value of BTOS if you are running PROGRESS on a BTOS/CTOS system.

SYNTAX

```
OPSYS
```

EXAMPLE

r-opsys.p

```
IF OPSYS = "unix" THEN UNIX ls.
ELSE IF OPSYS = "msdos" THEN DOS dir.
ELSE IF OPSYS = "os2" THEN OS2 dir.
ELSE IF OPSYS = "vms" then VMS directory.
ELSE IF OPSYS = "btos" then BTOS
              "[sys]<sys>files.run"
ELSE DISPLAY  files.
     OPSYS "is an unsupported operating system".
```

This procedure produces a listing of the files in your current directory. The OPSYS function determines which operating system you are running PROGRESS on, and uses the appropriate operating system command to produce the directory listing.

SEE ALSO DOS Statement, UNIX Statement, VMS Statement, BTOS Statement, CTOS Statement, OS2 Statement.

OR Function

Returns a TRUE value if either of two logical expressions is TRUE.

SYNTAX

expression OR *expression*

expression

A logical expression (a constant, field name, variable name or any combination of these whose value is logical, i.e. true/false, yes/no, value).

EXAMPLE

r-or.p

```
FOR EACH customer WHERE zip = 0 OR phone = "":
  DISPLAY cust-num name (COUNT) city st zip phone.
END.
```

This procedure lists customers who have no zip code (zip = 0) or that have no telephone number (phone = ""). It also displays how many customers are in the list.

SEE ALSO AND Operator, NOT Function

OS2 Statement

Runs a program, OS/2 command, OS/2 batch file, or starts the OS/2 command processor, allowing interactive processing of OS/2 commands.

SYNTAX

```
OS2 [SILENT] [ os2-command           [ argument                ] ... ]
             [ VALUE( expression )   [ VALUE( expression )     ]     ]
```

SILENT

After processing an OS2 statement, PROGRESS pauses, telling you to press the space bar to continue. When you press the space bar, PROGRESS clears the screen and continues processing. You can use the SILENT option to eliminate this pause. Use this option only if you are sure that the OS/2 program, command, or batch file will not generate output to the screen.

os2-command

The name of the program, command, or batch file you want to run. If you do not use this option, the OS2 statement starts the OS/2 command processor and remains at OS/2 level until you type "exit."

VALUE*(expression)*

expression in this option is an expression (a constant, field name, variable name, or any combination of these) whose value is the name of a program, command, or batch file you want OS/2 to execute.

argument

One or more arguments you want to pass to the program, command, or batch file being run by the OS2 statement. These arguments are expressions that PROGRESS converts to character values if necessary.

EXAMPLE

```
                                              r-os2.p
  IF OPSYS = "UNIX" then UNIX ls.
   ELSE IF OPSYS = "MSDOS" then DOS dir.
   ELSE IF OPSYS = "OS2" then OS2 dir.
   ELSE IF OPSYS = "VMS" then VMS directory.
   ELSE IF OPSYS = "BTOS" then  BTOS
   "[sys]<sys>files.run" files.
 ELSE DISPLAY OPSYS  "is an unsupported operating system".
```

If the operating system you are using is UNIX, this procedure runs the UNIX ls command. If the operating system is VMS, then the procedure runs the VMS directory command. If you are using DOS, this procedure runs the DOS dir command. If you are using OS/2, this procedure runs the OS2 dir command. If you are using BTOS then the [sys] < sys > files.run files command is executed. Otherwise, a message is displayed stating the operating system is unsupported.

NOTES

- If, for example, you use the OS2 statement in a procedure, the procedure will compile on, for example, a UNIX system, and the procedure will run as long as flow of control does not pass through the DOS statement while running on UNIX. You can use the OPSYS function to return the name of the operating system on which a procedure is being run. Using this function enables you to write applications that are fully transportable between any PROGRESS supported operating system even if they use the DOS, UNIX, etc. statements.
- You can also access the interactive OS/2 command processor by selecting option "e" from the main menu of PROGRESS Help.

SEE ALSO OPSYS Function, UNIX Statement, VMS Statement, BTOS Statement, CTOS Statement, DOS Statement.

OUTPUT CLOSE Statement

Closes the default output destination or the output stream you name with the STREAM keyword.

SYNTAX

```
OUTPUT [ STREAMstream ] CLOSE
```

STREAM *stream*

Specifies the name of a stream. If you do not name a stream, the unnamed stream is used. See the DEFINE STREAM reference page and Chapter 9 of the *Programming Handbook* for more information about streams.

EXAMPLE

r-out.p

```
    OUTPUT TO cust.dat.

    FOR EACH customer:
      DISPLAY cust-num name address address2 city
              st zip phone SKIP(2)
              WITH 1 COLUMN SIDE-LABELS.
    END.
 ➜  OUTPUT CLOSE.
    DISPLAY "Finished".
```

This procedure sends customer data to a file by using the OUTPUT TO statement. All statements that normally send output to the terminal send output to the file named cust.dat. After all customer data is written to the file, the OUTPUT CLOSE statement resets the output destination, usually the terminal. The final DISPLAY statement displays "Finished" on the terminal. To look at the contents of the cust.dat file, just press GET (F5) and type the name of the file.

NOTES

- The default output destination is whatever destination was active when the procedure began, usually the terminal unless the current procedure was called by another procedure while a different destination was active.

- A form feed (new page) is automatically output when a PAGED output stream is closed.

SEE ALSO DEFINE STREAM Statement, OUTPUT TO Statement, Chapter 9 of the *Programming Handbook.*

OUTPUT THROUGH Statement

Identifies a new output destination as the input to a UNIX process that PROGRESS will start.

SYNTAX

```
OUTPUT [ STREAM stream] THROUGH { program-name        }
                                { VALUE(expression )  }

                                [ argument            ] ...
                                [ VALUE( expression ) ]

                                [ PAGED                    ]
                                [ PAGE-SIZE(expression)    ]
                                [ ECHO                     ] ...
                                [ NO-ECHO                  ]
                                [ UNBUFFERED               ]
                                [ [NO-] MAP                ]
```

STREAM *stream*

Specifies the name of a stream. If you do not name a stream, PROGRESS uses the unnamed stream. See the DEFINE STREAM reference page and Chapter 9 of the *Programming Handbook* for more information about streams.

program-name

The name of the UNIX program to which you are supplying data from a procedure. This can be a standard UNIX command or your own program.

VALUE*(expression)*

An expression (a constant, field name, variable name, or any combination of these) whose value is the name of a UNIX program to which you are supplying data from a procedure.

argument

An argument you want to pass to the UNIX program. OUTPUT THROUGH passes this argument as a character string.

If the argument is the literal value "paged", "page-size", "echo", "no-echo", or "unbuffered" you must enclose it in quotes to prevent PROGRESS from using that argument as one of the PAGED, PAGE-SIZE, ECHO, NO-ECHO, or UNBUFFERED options for the OUTPUT THROUGH statement.

VALUE*(expression)*

An expression (a constant, field name, variable name, or any combination of these) whose value is the argument that you want to pass to the UNIX program. OUTPUT THROUGH passes the value of *expression* as a character string.

PAGED

Formats the output into pages.

PAGE-SIZE *expression*

Specifies the number of lines per page. The *expression* is a constant, field name, variable name, or any combination of these whose value is an integer. The default number of lines per page is 56. If you are using the TERMINAL option to direct output to the terminal, the default number of lines per page is the number of lines on the screen. If *n* is not 0 then PAGE-SIZE *n* implies PAGED. IF *n* is 0, the output is not paged.

ECHO

Sends all input data read from a file to the UNIX program. Data is echoed by default.

NO-ECHO

Suppresses the echoing of input data to the UNIX program.

[NO-] MAP *protermcap-entry*

The *protermcap-entry* is an entry from the PROTERMCAP file. Use MAP to send output to a device that requires different character mappings than those in effect for the current output stream. Typically, *protermcap-entry* is a slash-separated combination of a standard device entry and one or more language-specific add-on entries (MAP `laserwriter/french` or MAP `hp2/spanish/italian`, for example). PROGRESS uses the protermcap entries to build a translation table for the stream. Use NO-MAP to make PROGRESS bypass character translation altogether. See also Chapter 2 in the *Programming Handbook* for more information on PROTERMCAP and national language support.

UNBUFFERED

Writes one character at a time to a normally buffered data source, such as a file. You should use the UNBUFFERED option only when the output operations of a UNIX process invoked with the PROGRESS UNIX statement may be intermingled with the output being done by the PROGRESS statements that follow the OUTPUT THROUGH statement. That is, the OUTPUT THROUGH statement manages the buffering of output between the PROGRESS procedure the UNIX program that it invokes, but it does not handle the buffering of output to any other programs that the PROGRESS procedure may also invoke.

EXAMPLES

r-othru.p

```
OUTPUT THROUGH wc >wcdata.
  /* word count UNIX utility */

FOR EACH customer:
  DISPLAY name WITH NO-LABELS NO-BOX.
END.

OUTPUT CLOSE.
PAUSE 1 NO-MESSAGE.
UNIX cat wcdata.
UNIX SILENT rm wcdata.
```

In this example, the customer names are displayed and this output is sent as input to the UNIX wc (word count) command. The output of wc is directed to the file wcdata using the standard UNIX redirection symbol (>). Finally, the results are displayed as 3 integers representing the number of lines, words and characters that were in the data sent to wc.

r-othru2.p

```
OUTPUT THROUGH crypt mypass >ecust.

FOR EACH customer WHERE cust-num < 10:
  DISPLAY name WITH NO-LABELS NO-BOX.
END.

OUTPUT CLOSE.

UNIX crypt mypass <ecust.
```

The r–othru2.p procedure uses the UNIX "crypt" program which accepts lines of data, applies an algorithm based on an encryption key and writes the result to the UNIX standard output stream, which can be directed to a file. The output from the procedure is directed to crypt which encrypts the customer names based on the password "mypass". The results of the encryption are stored in the file ecust. Then, this file is decrypted and displayed.

NOTES

- When you use the OUTPUT CLOSE statement to close an output destination being used by an OUTPUT THROUGH statement, PROGRESS closes the pipe, waits one second and then continues.
- Since BTOS/CTOS, DOS, and VMS environments do not have an implementation of pipes, the PROGRESS command OUTPUT THROUGH, which is included to interact with piping, is not operational. However, if you use this statement in a PROGRESS procedure, the procedure will compile. In addition, the procedure will run as long as the flow of control does not pass through the unsupported statement.

SEE ALSO OUTPUT TO Statement, DEFINE STREAM Statement, Chapter 9 of the *Programming Handbook*

OUTPUT TO Statement

Specifies a new output destination.

SYNTAX

```
OUTPUT [ STREAM stream ] TO

  { PRINTER                }  [ PAGED                        ]
  { opsys-file             }  [ PAGE-SIZE expression         ]
  { opsys-device           }  [ APPEND                       ] ...
  { TERMINAL               }  [ ECHO                         ]
  { VALUE( expression )    }  [ NO-ECHO                      ]
                              [ UNBUFFERED                   ]
                              [ [NO-] MAP protermcap-entry   ]
```

STREAM *stream*

Specifies the name of a stream. If you do not name a stream, the unnamed stream is used. See the DEFINE STREAM reference page and Chapter 9 of the *Programming Handbook* for more information about streams.

PRINTER

Sends output to the printer. When you use this option, it implies that the device you are sending output to is paged, unless you also specify PAGE-SIZE 0. On UNIX systems, the printer spooling facilities (lp or lpr) are used automatically.

opsys-file

The name of an ASCII file to which you want to direct output from a procedure.

opsys-device

The name of a operating system device.

TERMINAL

Indicates that you want to direct output to the terminal. The terminal is the default output destination.

VALUE(*expression*)

An expression (a constant, field name, variable name, or any combination of these) whose value is the destination to which you want to send data.

PAGED

Formats the output into pages. Form feeds are represented by the CTRL-L character. When output is paged a page break occurs every 56 lines. PAGED is automatic for output to a printer.

[NO-] MAP *protermcap-entry*

The *protermcap-entry* is an entry from the PROTERMCAP file. Use MAP to send output to a device that requires different character mappings than those in effect for the current output stream. Typically, *protermcap-entry* is a slash-separated combination of a standard device entry and one or more language-specific add-on entries (MAP `laserwriter/french` or MAP `hp2/spanish/italian`, for example). PROGRESS uses the protermcap entries to build a translation table for the stream. Use NO-MAP to make PROGRESS bypass character translation altogether. See also Chapter 2 in the *Programming Handbook* for more information on PROTERMCAP and national language support.

PAGE-SIZE *expression*

Specifies the number of lines per page. The *expression* is a constant, field name, variable name, or any combination of these whose value is an integer. The default number of lines per page is 56. If you are using the TERMINAL option to direct output to the terminal, the default number of lines per page is the number of lines on the screen. If *expression* is not 0, then PAGE-SIZE *expression* implies that the output is PAGED. If *expression* is 0, the output is not PAGED.

APPEND

Appends the output to the end of a file.

ECHO

Sends all input data read from a file to the output destination. Data is echoed by default.

NO-ECHO

Suppresses the echoing of input data to the output destination.

UNBUFFERED

Writes one character at a time to a normally buffered data source, such as a file. You should use the UNBUFFERED option only when the output operations of a UNIX or VMS process invoked with the PROGRESS UNIX or VMS statement may be intermingled with the output being done by the PROGRESS statements that follow the OUTPUT TO statement.

EXAMPLES

```
                                                      r-out.p

➡ OUTPUT TO cust.dat.

  FOR EACH customer:
    DISPLAY cust-num name address address2 city st zip
            phone SKIP (2) WITH 1 COLUMN SIDE-LABELS.
  END.
  OUTPUT CLOSE.
  DISPLAY "Finished".
```

The r-out.p procedure sends customer data to a file. The OUTPUT TO statement directs subsequent output to a file, so all statements that normally send output to the terminal send output to that file. After all the customer data has been displayed to the file, the OUTPUT CLOSE statement resets the output destination to its previous state, usually the terminal. The final DISPLAY statement displays “Finished” on the terminal because that is the new output destination.

```
                                                    r-termpg.p

➡ OUTPUT TO TERMINAL PAGED.
  DEFINE VAR x AS INTEGER.

  FOR EACH customer BREAK BY sales-rep:
    FIND salesrep OF customer.

    FORM HEADER TODAY
      "Customer Listing For " to 43
      "Page " to 55 PAGE-NUMBER - x TO 58 FORMAT "99"
      (salesrep.slsname) FORMAT "x(30)" AT 25
    WITH FRAME hdr PAGE-TOP CENTERED.

    VIEW FRAME hdr.
    DISPLAY cust-num COLUMN-LABEL "Customer!Number"
            name LABEL "Name"
            phone COLUMN-LABEL "Phone!Number" WITH CENTERED.
    IF LAST-OF (cust.sales-rep)
    THEN DO:
      x = PAGE-NUMBER.
      PAGE.
    END.
  END.
```

The r-termpg.p procedure sends customer data to the terminal. The OUTPUT TO TERMINAL PAGED statement directs output to the terminal in a paged format; all statements send output to the terminal one page at a time.

NOTES

- OUTPUT TO TERMINAL is the default unless the procedure was called by another procedure while a different output destination was active. The output destination at the beginning of the procedure is the current output destination of the calling procedure.
- OUTPUT TO TERMINAL PAGED clears the screen and displays output on scrolling pages the length of the screen. PROGRESS pauses before each page header. You can alter the pause using the PAUSE statement.
- PROGRESS can display paged output to the terminal for frames that are wider than the width of the screen. The output is wrapped.
- If you send data to a file and you plan to use that data file later as input to a procedure, consider using the EXPORT statement. See the INPUT FROM statement for more information.
- If you send output to a device other than the terminal, ROW options in Frame phrases have no effect. ROW options also have no effect when you send output to a PAGED terminal. In cases where you do not use the NO-BOX option with a Frame phrase, PROGRESS omits the bottom line of the box, converts the top line to blanks, and ignores the sides of the box.
- All messages, including compiler error messages and messages produced by the MESSAGE statement, are sent to the current output destination.
- Under UNIX, if you want to use a print spooler and specify spooler options you can use OUTPUT TO PRINTER if you specified the spooler options in the Specify Printer (-o) option. (See Chapter 3 of *System Administration II: General* for more information on start-up options.) Alternatively, you can use either one of the following statements:

```
OUTPUT THROUGH lp spooler-options PAGED.
```

```
OUTPUT THROUGH lpr spooler-options PAGED.
```

- Under VMS, you can specify a print queue using the /PRINTER start-up option.
- Under DOS, PROGRESS provides special error checking services in support of OUTPUT TO PRINTER (e.g. out of paper on the printer) as follows:
  - When running interactively, output is sent to the first parallel printer port (PRN or LPT1) regardless of the setting of the DOS MODE command.
  - When running in batch, output is sent to the parallel or serial port indicated by the MODE command. Normal DOS error handling applies.
  - If you use the Specify Printer (-o) option, normal DOS error handling applies even if you specify PRN or LPT1.
- Under DOS, OUTPUT TO LPT1, LPT2, PRN, etc. bypasses PROGRESS' special handling of the parallel printer. You can redirect output with the DOS MODE command. Be sure to use the PAGED option.
- You must use a printer control sequence to change the number of lines per page produced by your printer.
- Under BTOS/CTOS, you can use the "Printer Name" option to specify a printer. Both direct and spooled printers are supported. for example, "[SPL]" , "[PTR]B" , "[LPT]" are all valid names.

SEE ALSO DEFINE STREAM Statement, OUTPUT CLOSE Statement, PAGE-SIZE Function, Chapter 9 of the *Programming Handbook*

OVERLAY Function

Overlays a character expression in a field or variable starting at a given position, and optionally for a given length.

SYNTAX

```
OVERLAY(target, position [ , length ] )  = expression
```

target

The name of the character field or variable in which you want to overlay an *expression.*

position

An integer expression (a constant, field name, variable name, or any combination of these that results in an integer value) that indicates the first character position in *target* at which you want to store *expression.* If *position* is longer than *target,* PROGRESS pads *target* with blanks to equal the length of *position.*

length

An integer expression (a constant, field name, variable name, or any combination of these that results in an integer value) that indicates the number of positions you want to allocate for the storage of *expression. expression* is truncated or padded with blanks to match *length.* If you do not use the *length* argument, OVERLAY uses the entire *expression.*

expression

A constant, field name, variable name, or any combination of these that results in a character string that you want to overlay on *target.* If you specify *length,* the *expression* is truncated or padded with blanks to match *length.*

EXAMPLE

The r-replc1.p procedure allows you to search for, and replace text strings in a paragraph on your screen. When you run the procedure, you see the paragraph, which is an array with an extent of 5. You also see a frame that prompts you for a text string to search for, and text with which to replace the string.

The procedure searches the paragraph, one line at a time, for the text you enter. The procedure uses the OVERLAY function to replace the string of old text with the string of new text. The procedure also does other work to determine the length of the old text and the new text that you enter.

r-replc1.p

```
DEFINE VARIABLE chktext  AS CHARACTER.
DEFINE VARIABLE i        AS INTEGER.
DEFINE VARIABLE chkndx   AS INTEGER.
DEFINE VARIABLE ndx      AS INTEGER.
DEFINE VARIABLE old-text AS CHARACTER.
DEFINE VARIABLE new-text AS CHARACTER.
DEFINE VARIABLE max-len  AS INTEGER.
DEFINE VARIABLE comment  AS CHARACTER FORMAT "x(49)"
       EXTENT 5 INITIAL
       ["You are probably interested in PROGRESS because",
        "you have a lot of information to organize.  You",
        "want to get at the information, add to it, and",
        "change it, without a lot of work and aggravation.",
        "You made the right choice with PROGRESS." ].

DISPLAY comment WITH CENTERED FRAME comm NO-LABELS
         TITLE "Why You Chose PROGRESS" ROW 4.

REPEAT:
  SET old-text LABEL "Enter text to search for"
      new-text LABEL "Enter text to replace with"
  WITH FRAME replace SIDE-LABELS CENTERED.
  max-len = MAXIMUM(LENGTH(old-text), LENGTH(new-text)).
  DO i = 1 TO 5:
    ndx = 1.
    DO ndx = 1 TO LENGTH(comment[i]):
      chktext = SUBSTRING(comment[i], ndx).
      chkndx = INDEX(chktext, old-text).
      IF chkndx <> 0 THEN DO:
        ndx = ndx + chkndx - 1.
        OVERLAY(comment[i], ndx, max-len) = new-text.
        ndx = max-len.
      END.
    END.
    DISPLAY comment[i] WITH FRAME comm.
  END.
END.
```

SEE ALSO SUBSTRING Function

PAGE Statement

Starts a new output page for PAGED output. No action is taken if output is already positioned at the beginning of a page.

SYNTAX

```
PAGE [ STREAM stream ]
```

STREAM *stream*
Specifies the name of a stream. If you do not name a stream, the unnamed stream is used. See the DEFINE STREAM reference page and Chapter 9 of the *Programming Handbook* for more information about streams.

EXAMPLE

r-page.p

```
DEFINE VARIABLE laststate AS CHARACTER.

OUTPUT TO PRINTER.
FOR EACH customer BY st:
    IF st <> laststate THEN DO:
    IF laststate <> "" THEN PAGE.
    laststate = st.
  END.
  DISPLAY cust-num name address city st.
END.
```

This procedure prints a customer report, categorized by state, and starts a new page for each state.

NOTES

- If the current output destination is not a paged device (you did not use the PAGED option in the OUTPUT statement), the PAGE statement has no effect.
- PAGE has no effect if you are already at the top of a new page.
- If any PAGE-TOP or PAGE-BOTTOM frames are active, these are output prior to the next display.

SEE ALSO DEFINE STREAM Statement, OUTPUT TO Statement, Chapter 9 of the *Programming Handbook*

PAGE-NUMBER Function

Returns the page number of the output destination. If the output stream is not paged, PAGE-NUMBER returns a value of 0.

SYNTAX

```
PAGE-NUMBER [ (stream ) ]
```

stream

The name of an output stream. If you do not name a stream, PAGE-NUMBER returns the page number of the default unnamed output stream.

EXAMPLE

r-pgnbr.p

```
OUTPUT TO PRINTER.
FOR EACH customer:
  FORM HEADER "Customer report" AT 30
              "Page:" AT 60 PAGE-NUMBER
              FORMAT ">>9" SKIP(1).
  DISPLAY cust-num name address city st.
END.
```

This procedure prints a customer report with the page number on each page of the report.

SEE ALSO OUTPUT TO Statement, PAGE Statement

PAGE-SIZE Function

Returns the page size (lines per page) of an output destination. If the output stream is not paged, PAGE-SIZE returns a value of 0.

SYNTAX

```
PAGE-SIZE [ ( stream ) ]
```

stream

The name of an output stream. If you do not name a stream, PAGE-SIZE returns the page size of the default unnamed output stream.

EXAMPLE

r-pgsize.p

```
OUTPUT TO PRINTER.
FOR EACH customer BREAK BY st:
  DISPLAY cust-num name address city st.
  IF LAST-OF(st) THEN DO:
    IF LINE-COUNTER + 4 > PAGE-SIZE
    THEN PAGE.
    ELSE DOWN 1.
  END.
END.
```

This procedure prints a customer report categorized by state. At the end of each state category, it tests to see whether there are at least four lines left on the page. The LINE-COUNTER function returns the current line number of output. If that number plus 4 is greater than the total number of lines on the page (returned by the PAGE-SIZE function), then the procedure skips to a new page. If there are four or more lines left, the procedure skips a line before printing the next customer record.

SEE ALSO OUTPUT THROUGH Statement, OUTPUT TO Statement

PAUSE Statement

Suspends processing indefinitely, or for a specified number of seconds, or until the user presses any key.

SYNTAX

```
PAUSE [ n ] [ BEFORE-HIDE ] [ MESSAGE message ]
                            [ NO-MESSAGE      ]
```

n

The number of seconds for which you want to suspend processing. If you do not use this option, PROGRESS suspends processing until you press any key.

BEFORE-HIDE

Specifies the pause action to be taken whenever frames are hidden automatically. If you specify *n*, *n* is the number of seconds to pause before hiding. If you do not specify *n*, then the pause lasts until you press any key.

MESSAGE *message*

When PROGRESS encounters a PAUSE statement, it displays the message "Press space bar to continue" on the status line of the terminal screen. You use the MESSAGE option to override that default message. *message* is a constant character string.

NO-MESSAGE

Tells PROGRESS to pause but not to display a message on the status line of the terminal screen.

EXAMPLE

r-pause.p

```
PAUSE 2 BEFORE-HIDE MESSAGE
  "Pausing 2 seconds".
FOR EACH customer WITH 13 DOWN:
  DISPLAY cust-num name.
END.
```

The FOR EACH block in this procedure reads each of the records from the customer file and displays information from each record. Because the DISPLAY uses a down frame (multiple records displayed in the frame), PROGRESS normally fills the screen with as many records as possible and then displays the message "Press space bar to continue". The PAUSE 2 BEFORE-HIDE message tells PROGRESS to pause only 2 seconds before hiding the frame.

NOTES

- After you use a PAUSE, that statement is in effect for all the procedures run in that session unless it is overridden by other PAUSE statements in those procedures, or until you return to the editor.
- Using the PAUSE *n* BEFORE-HIDE statement is a good way to write a demonstration application that runs by itself.
- PROGRESS automatically pauses before removing a frame from the screen, displaying the "Press space bar to continue" message if you have not had a chance to see the data in the frame.
- When a PAUSE occurs, PROGRESS clears any keystrokes buffered from the keyboard, discarding any "type-ahead" characters.

PDBNAME Function

The PDBNAME function returns the physical name of a currently connected database.

SYNTAX

$$\text{PDBNAME} \left\{ \begin{array}{l} (\textit{integer-expression}) \\ (\textit{logical-name}) \\ (\textit{alias}) \end{array} \right\}$$

integer-expression

If the parameter supplied to PDBNAME is an integer expression, and there are, for example, three currently connected databases, then `PDBNAME(1)`, `PDBNAME(2)`, and `PDBNAME(3)` return their physical names. Also, continuing the same example of three connected databases, `PDBNAME(4)`, `PDBNAME(5)`, etc., return the ? value.

logical-name or *alias*

This form of the PDBNAME function requires a quoted character string or a character expression as a parameter. If the parameter is the logical name of a connected database or an alias of a connected database, then the physical name is returned. Otherwise, the ? value is returned.

EXAMPLE

r-pdbnml.p

```
IF SEARCH(PDBNAME(1) + ".db") = ? THEN DO:
  BELL.
  MESSAGE
    "Your database has been deleted, so this session will
     now terminate.".
  PAUSE.
  QUIT.
END.
```

This procedure checks to see if your first connected PROGRESS database still exists. If it doesn't exist, your session terminates. This procedure could be useful if you are working on a system where your session would normally continue after the database was deleted.

NOTES

- The old DBNAME function has been retained for compatibility and is equivalent to PDBNAME(1).

SEE ALSO CONNECT, DISCONNECT, CREATE ALIAS, and DELETE ALIAS statements; CONNECTED, LDBNAME, SDBNAME DBTYPE, DBRESTRICTIONS, GATEWAYS, FRAME-DB, and NUM-DBS functions.

PROGRAM-NAME Function

Returns the name of the calling program.

SYNTAX

```
PROGRAM-NAME( n)
```

n

The numeric argument. If $n = 1$, the name of the current program is returned. If $n = 2$, the name of the calling program is returned. If there is no calling program, for example PROGRAM-NAME is run from the edit buffer, then a ? is returned.

EXAMPLE

r-prgnm.p

```
/* Note this program should be run as a subroutine */
/* the deeper the nesting, the better the illustration */

 DEFINE VAR level AS INT INITIAL 1.
 REPEAT WHILE PROGRAM-NAME(level) <> ?.
 DISPLAY LEVEL PROGRAM-NAME(level) FORMAT "x(30)".
 level = level + 1.
 END.
```

This procedure returns the names of any procedure(s) that called it, and how many levels deep the procedure was nested.

NOTES

- The PROGRAM-NAME function is especially useful when developing online help. For example, if your `applhelp.p` file contains :

```
/* applhelp.p */

DEF VAR i                    AS INT.
DEF VAR plist                AS CHAR FORMAT "x(60)".

FORM plist
WITH FRAME what-prog OVERLAY
    ROW 10 CENTERED 5 DOWN NO-LABELS
    TITLE " Program Trace ".

i = 2.
DO WHILE PROGRAM-NAME(i) <> ? :
   IF i = 2 THEN
      plist = "Currently In        : " + PROGRAM-NAME(i).
   /* see note below */
   ELSE
      plist = "Which was called by: " + PROGRAM-NAME(i).
  /* see note below */
   i = i + 1.
   DISPLAY plist WITH FRAME what-prog.
   DOWN WITH FRAME what-prog.
END.
PAUSE.
HIDE FRAME what-prog.

/*
   To display the full path name of the
   program trace you can use the SEARCH
   function.  Just replace PROGRAM-NAME(i)
   with SEARCH(PROGRAM-NAME(i)).
*/
```

and you press HELP, you'll receive a program-trace.

PROGRESS Function

Returns one of the following character values that identifies the PROGRESS product that is running: Full, Query, Run-Time. Can also return COMPILE if you are using the developer's toolkit, or COMPILE-ENCRYPT if you are using the run-time compiler.

SYNTAX

```
PROGRESS
```

EXAMPLES

r-progfn.p

```
DISPLAY "You are currently running this PROGRESS product:  "
         PROGRESS.
```

r-progfn.p

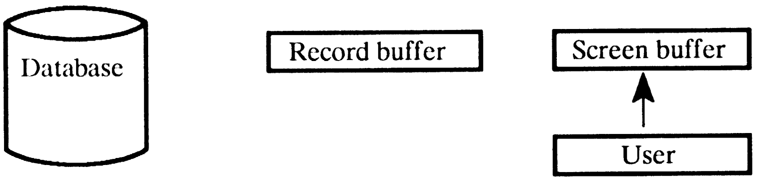
```
/* Depending on the version of Progress you're running, */
/* the main menu reflects available features for end-user  */
   DEFINE VARIABLE menu AS CHARACTER EXTENT 3
    DEFINE VARIABLE exit-prompt AS CHARACTER.

   IF PROGRESS EQ "FULL" THEN
       exit-prompt = "  3. Return to Full Editor ".
   ELSE IF PROGRESS EQ "QUERY" THEN
       exit-prompt = "  3. Return to Query Editor ".
   ELSE IF PROGRESS EQ "RUN-TIME" THEN
       exit-prompt = "  3. Exit Program".

   DO WHILE TRUE:
    DISPLAY
      "  1. Display Customer Data" @ menu[1] SKIP
      "  2. Display Order Data"    @ menu[2] SKIP
         exit-prompt               @ menu[3]
            FORMAT "x(26)" SKIP
      WITH FRAME choices NO-LABELS.

     CHOOSE FIELD menu AUTO-RETURN WITH FRAME choices
        TITLE "Demonstration Menu" CENTERED ROW 10.
      HIDE FRAME choices.
      IF FRAME-INDEX EQ 1 THEN message
            "You picked option 1.".
      ELSE IF FRAME-INDEX EQ 2 THEN message
            "You picked option 2.".
      ELSE IF FRAME-INDEX EQ 3 THEN RETURN.
   END.
```

This example uses the PROGRESS function to determine which exit prompt will display on a menu.

PROMPT-FOR Statement

Requests input and places that input in the screen buffer (frame).

DATA MOVEMENT



SYNTAX

```
PROMPT-FOR [ STREAM stream ]
              ┌ field [ format-phrase ] [ WHEN expression ]        ┐
              │ TEXT (field [format-phrase] ... )                  │
              │ constant  [AT n]                                    │
              │           [TO n]                                    │ ...
              │ ^                                                   │
              │ SPACE[( n )]                                        │
              └ SKIP[( n )]                                         ┘
              [ GO-ON ( key-label ... ) ]
              [ frame-phrase ]
              [ EDITING-phrase ]
```

```
PROMPT-FOR [ STREAM stream] record [EXCEPT field ...] [ frame-phrase ]
```

STREAM *stream*

Specifies the name of a stream. If you do not name a stream, PROGRESS uses the unnamed stream. See the DEFINE STREAM reference page and Chapter 9 of the *Programming Handbook* for more information about streams.

field

The name of the field or variable whose value you want to enter and store in the screen buffer. Remember that the PROMPT-FOR statement only accepts input and stores it in the screen buffer. The underlying record buffer of a field or variable is unaffected.

For example:

```
DEFINE VARIABLE x AS INTEGER.
PROMPT-FOR X.
DISPLAY X.
```

This procedure prompts for the value of the variable x and stores that value in the screen buffer. The DISPLAY statement displays the value of the x variable that is stored in the record buffer. The value of the variable is not affected by the PROMPT-FOR statement.

In the case of array fields, array elements with constant subscripts are treated just like any other field. Array fields with no subscripts or in the FORM statement are expanded as though you had typed in the implicit elements. Array fields with expressions as subscripts are handled as described on the DISPLAY statement reference page.

format-phrase

Specifies one or more frame attributes for a field, variable, or expression. Here is the syntax for *format-phrase*:

```
[  AT n                                            ]
[  AS datatype                                     ]
[  ATTR-SPACE                                      ]
[  AUTO-RETURN                                     ]
[  BLANK                                           ]
[  COLON n                                         ]
[  COLUMN-LABEL   label   [ ! label ]...           ]  ...
[  DEBLANK                                         ]
[  FORMAT string                                   ]
[  HELP string                                     ]
[  LABEL string                                    ]
[  LIKE field                                      ]
[  NO-ATTR-SPACE                                   ]
[  NO-LABEL                                        ]
[  TO n                                            ]
[  VALIDATE ( condition,msg-expression)            ]
```

For more information on *format-phrase*, see the Format Phrase reference page.

WHEN *expression*

Prompts for the field only when *expression* has a value of TRUE. Here, *expression* is a field name, variable name, or any combination of these whose value is logical.

TEXT

Defines a group of character fields or variables (including array elements) to use automatic word-wrap. The TEXT option works only with character fields. When you insert data in the middle of a TEXT field, PROGRESS wraps data that follows into the next TEXT field, if necessary. If you delete data from the middle of a TEXT field, PROGRESS wraps data that follows up into the empty area.

If you enter more characters than the format for the field allows, PROGRESS discards the extra characters. The character fields must have formats of the form "x(n)". A blank in the first column of a line marks the beginning of a paragraph. Lines within a paragraph are treated as a group and will not wrap into other paragraphs.

The following table lists the keys you can use within a TEXT field and their actions.

Table 22: Key Actions in a TEXT Field

KEY	ACTION
APPEND-LINE	Combines the line the cursor is on with the next line.
BACK-TAB	Moves the cursor to the previous TEXT field.
BREAK-LINE	Breaks the current line into two lines beginning with the character the cursor is on.
BACK SPACE	Moves the cursor one position to the left and deletes the character at that position. If the cursor is at the beginning of a line, BACK SPACE moves the cursor to the end of the previous line.
CLEAR	Clears the current field and all fields in the TEXT group that follow.
DELETE-LINE	Deletes the line the cursor is on.
NEW-LINE	Inserts a blank line below the line the cursor is on.
RECALL	Clears fields in the TEXT group and returns initial data values for the group.
RETURN	If you are in overstrike mode, moves to the next field in the TEXT group on the screen. If you are in insert mode, the line breaks at the cursor and the cursor is positioned at the beginning of the new line.
TAB	Moves to the field after the TEXT group on the screen. If there is no other field, the cursor moves to the beginning of the TEXT group.

In the procedure that follows, the s-com, or "Order Comments" field is a TEXT field. Run the procedure and enter text in the field to see how the TEXT option works.

r-text.p

```
DEFINE VARIABLE s-com AS CHARACTER FORMAT "x(40)" EXTENT 5.

FORM "Shipped  :"      order.sdate AT 13 SKIP
     "Misc Info:"      order.misc-info AT 13 SKIP(1)
     "Order Comments:" s-com AT 1
WITH FRAME o-com CENTERED NO-LABELS
     TITLE "Shipping Information".

FOR EACH customer, EACH order OF customer:
  DISPLAY cust.cust-num cust.name order.order-num
          order.odate order.pdate
          WITH FRAME order-hdr CENTERED.
  UPDATE sdate misc-info TEXT(s-com) WITH FRAME o-com.
  s-com = "".
END.
```

constant AT *n*

A constant (literal) value that you want displayed in the frame; *n* is the column at which you want to start the display.

constant TO *n*

A constant value that you want displayed in the frame. *n* is the column at which you want to end the display.

^

You use the caret to tell PROGRESS to ignore an input field when input is being read from a file. Also, the statement

```
PROMPT-FOR ^
```

will read a line from an input file and ignore that line. This is an efficient way of skipping over lines.

SPACE (*n*)

Identifies the number (*n*) of blank spaces to be inserted after the expression is displayed. *n* can be zero (0). If the number of spaces you specify is more than the spaces left on the current line of the frame, a new line is started and any extra spaces discarded. If you do not use this option or do not use *n*, one space is inserted between items in the frame.

SKIP (*n*)

Identifies the number (*n*) of blank lines to be inserted after the expression is displayed. *n* can be zero (0). If you do not use this option, PROGRESS does not skip a line between expressions unless the expressions do not fit on one line. If you use the SKIP option but do not specify *n*, or if *n* is 0, PROGRESS starts a new line unless it is already at the beginning of a new line.

GO-ON(*key*...)

key... is a list of keyboard key labels. The GO-ON option tells PROGRESS to take the GO action when the user presses any of the keys listed in *key*... . The keys you list are used in addition to keys that perform the GO action by default (such as F1 or RETURN on the last field) or because of ON statements.

When you list a key in *key*..., you use the keyboard label of that key. For example, if your keyboard has an F2 key and you want PROGRESS to take the GO action when the user presses that key, you use the statement GO-ON(F2). If you list more than one key, separate those keys by spaces, not commas. See Chapter 2 of the *Programming Handbook* for a list of keys.

frame phrase

Specifies the overall layout and processing properties of a frame. Here is the syntax for *frame phrase*:

```
       [  ACCUM                                                   ]
       [  ATTR-SPACE                                              ]
       [  CENTERED                                                ]
       [        { [ DISPLAY ] color-phrase  }                     ]
       [  COLOR { PROMPT color-phrase       } ...                 ]
       [  COLUMN  expression                                      ]
       [  n COLUMNS                                               ]
       [  DOWN                                                    ]
       [  expression     DOWN                                     ]
       [  FRAME frame                                             ]
WITH   [  NO-ATTR-SPACE                                           ]
       [  NO-BOX                                                  ]  ...
       [  NO-HIDE                                                 ]
       [  NO-LABELS                                               ]
       [  NO-UNDERLINE                                            ]
       [  NO-VALIDATE                                             ]
       [  OVERLAY                                                 ]
       [  PAGE-BOTTOM                                             ]
       [  PAGE-TOP                                                ]
       [  RETAIN n                                                ]
       [  ROW  expression                                         ]
       [  SCROLL  n                                               ]
       [  SIDE-LABELS                                             ]
       [  TITLE [ COLOR   color-phrase      ] expression          ]
       [  TOP-ONLY                                                ]
       [  WIDTH n                                                 ]
```

For more information on *frame-phrase*, see the Frame Phrase reference page.

EDITING-*phrase*

Identifies processing to take place as each keystroke is entered.

```
[ label :] EDITING: statement...     END.
```

For more information on EDITING-*phrase*, see the EDITING Phrase reference page.

record

The name of a record buffer. All of the fields in the record will be processed exactly as if you had prompted for each of them individually.

To use PROMPT-FOR with a record in a file defined for multiple databases, you must qualify the record's filename with the database name. See the description of the Record Phrase for more information.

EXCEPT *field*
All fields except those fields listed in the EXCEPT phrase are affected.

EXAMPLES

r-prmpt.p

```
    REPEAT:
→     PROMPT-FOR customer.cust-num.
      FIND customer USING cust-num NO-ERROR.
      IF NOT AVAILABLE customer THEN DO:
        MESSAGE "No such customer number.".
        UNDO, RETRY.
      END.
      DISPLAY name phone sales-region.
    END.
```

The r-prmpt.p procedure requests a customer number from the user and stores that number in the screen buffer. The FIND statement reads a record from the customer database file.

r-prmpt2.p

```
    REPEAT:
→     PROMPT-FOR salesrep.sales-rep
        LABEL "Sales rep's initials"
        WITH FRAME namefr ROW 2 SIDE-LABELS.
      FIND salesrep USING sales-rep.
      DISPLAY slsname slstitle slsquota
        WITH 1 DOWN NO-HIDE.
    END.
```

The r-prmpt2.p procedure requests the initials of a sales representative and stores those initials in the screen buffer. The FIND statement uses the initials stored in the screen buffer to read a record from the salesrep database file. After finding the record, the procedure displays some sales rep information.

NOTES

- PROMPT-FOR puts user-supplied data into a screen buffer. It does not put any data into a record buffer. Therefore, if you want to use the data in the screen buffer, you must use either the INPUT function to refer to the data in the screen buffer or the ASSIGN statement to move the data from the screen buffer into a record buffer. You can also use the USING option to FIND a record with the screen data index value.

- When PROGRESS compiles a procedure, it designs all the frames used by that procedure. When it encounters a PROMPT-FOR statement, PROGRESS designs the display of the fields being prompted for. When the procedure is run, the PROMPT-FOR statement puts data into those fields.

- If you are getting input from a device other than the terminal, and the number of characters read by the PROMPT-FOR statement for a particular field or variable exceeds the display format for that field or variable, PROGRESS returns an error. However, if you are setting a logical field that has a format of "y/n" and the data file contains a value of "yes" or "no", PROGRESS converts that value to "y" or "n".

SEE ALSO DEFINE Stream Statement, EDITING Phrase, Format Phrase, Frame Phrase

PROPATH Statement

Sets the PROPATH environment variable for the current PROGRESS session.

SYNTAX

```
PROPATH = string-expression
```

string-expression
A field, variable, string constant, or combination of these that evaluates to a character string. The character string should be a list of directory paths. The directory names in the path can be separated either by commas or by the appropriate separation character for your operating system. The directory pathnames themselves can use either the UNIX format for pathnames (`/dir1/dir2/dir3`, for example) or the standard pathname format for your operating system. Use the slash-separated directory name format if you are concerned about portability across multiple operation systems.

EXAMPLES

r-ppath.p

```
DEFINE VARIABLE menu AS CHARACTER EXTENT 4 FORMAT "X(20)"
   INITIAL ["1. Sales","2. Acctg","3. Personnel","4. Exit"].
DEFINE VARIABLE proglist AS CHARACTER EXTENT 4 FORMAT "X(8)"
   INITIAL ["sales.p","acctg.p","per.p","exit.p"].
DEFINE VARIABLE ppath AS CHARACTER EXTENT 4 INITIAL
   ["sales/s-procs","acctg/a-procs","per/p-procs",",","].

REPEAT:
  DISPLAY menu WITH TITLE " M A I N   M E N U " CENTERED
     1 COLUMN 1 DOWN NO-LABELS ROW 8 ATTR-SPACE.
  CHOOSE FIELD menu AUTO-RETURN.
  HIDE.
➡PROPATH = ppath[FRAME-INDEX].
  RUN VALUE(proglist[FRAME-INDEX]).
END.
```

The `r-ppath.p` procedure displays a strip menu with four choices. The procedure defines three arrays: `menu` holds the items for selection on the menu, `proglist` holds the names of the programs associated with the menu selections, and `ppath` holds the appropriate PROPATHs for each program. The CHOOSE statement allows the user to select an item from the strip menu.

PROGRESS uses the menu selectionnumber as an index into the `ppath` and `proglist` arrays. PROGRESS sets the PROPATH and runs the program.

Another simple example that changes and displays the PROPATH:

```
                                              r-prpath.p

PROPATH=ENTRY(1,PROPATH) +
      ",/dlc,/dlc/proguide,/dlc/appl1/procs".
DISPLAY PROPATH.
```

NOTES

- Changes to PROPATH last only for the current session. Any subprocesses inherit the PROPATH in effect when the PROGRESS session started.
- When you start PROGRESS, the DLC, PRODEMO, and PROGUIDE directories are automatically added to your PROPATH. If you use the PROPATH statement to change the PROPATH, these directories are lost unless you explicitly include them.
- PROGRESS replaces separation characters in *expression* (":" for UNIX and BTOS, ";" for DOS and OS/2, and "." for VMS) with commas, so the resulting PROPATH string can be accessed with the ENTRY function. Therefore, file pathnames passed in *expression* must not include embedded commas.

SEE ALSO PROPATH Function, ENTRY function, *System Administration I: Environments* (for a discussion of environment variables on various operating systems)

PROPATH Function

Returns the current value of the PROPATH environment variable.

SYNTAX

```
PROPATH
```

EXAMPLE

```
r-ppath1.p

➨ DISPLAY PROPATH.

  DEFINE VARIABLE i AS INTEGER.
  REPEAT i=1 TO NUM-ENTRIES(PROPATH):
➨   DISPLAY ENTRY(i,PROPATH).
  END.
```

This procedure first displays a comma-separated list of the directories in the current PROPATH. It then displays each directory in the current PROPATH, one per line.

NOTES

- PROGRESS stores the PROPATH as a comma-separated list of directories. That is, PROGRESS strips the operating-specific separation characters (":" in UNIX and BTOS, ";" in DOS and OS/2, and "." for VMS) and replaces them with commas.

- The default format for PROPATH is "x(70)".

SEE ALSO PROPATH Statement

PUT Statement

Sends the value of one or more expressions to an output destination other than the terminal.

SYNTAX

```
PUT [ STREAM stream ] [ UNFORMATTED ]
    [ expression  [ FORMAT string  ]            ]
    [             [ AT expression  ] ...        ]
    [             [ TO expression  ]            ] ...
    [ SKIP[( expression )]                      ]
    [ SPACE[( expression )]                     ]
```

```
PUT [ STREAM stream ] CONTROL expression ...
```

STREAM *name*
: Specifies the name of a stream. If you do not name a stream, PROGRESS uses the unnamed stream. See the DEFINE STREAM reference page and Chapter 9 of the *Programming Handbook* for more information about streams.

UNFORMATTED
: Tells PROGRESS to display each expression in the same format as that produced by the EXPORT statement, but without quotes.

CONTROL *expression*
: *expression* is a control sequence that you want to send without affecting the current line, page counters, and positions maintained within PROGRESS. Following CONTROL, *expression* can include null character constants of the form NULL or NULL(*expression*), where *expression* is specifies the number of NULLs to be sent. See NOTES.

expression
: A constant, field name, variable name, or any combination of these.

FORMAT *string*
: The format in which you want to display the *expression*. If you do not use the FORMAT option, PROGRESS uses the defaults shown in Table 23.

Table 23: Default Display Formats

Type of Expression	Default Format
Field	Format from Dictionary
Variable	Format from variable definition
Constant character	Length of character string
Other	Default format for the data type of the expression. Table 24 shows these default formats.

Table 24: Default Data Type Display Formats

Data Type of Expression	Default Format
Character	x(8)
Date	99/99/99
Decimal	->>,>>9.99
Integer	->,>>>,>>9
Logical	yes/no

AT *expression*

Specifies the column position at which you want to place the value being output. If that position has already been used on the current line, PUT skips to the next line and puts the *expression* in the specified column.

TO *expression*

Specifies the column position at which you want to end the value being output. If that position has already been used on the current line, PUT skips to the next line and puts the *expression* in the specified column.

SKIP(*expression*)

Specifies the number of new lines you want to output. If you do not use the SKIP option, PUT will not start a new line to the output stream. If you use the SKIP parameter but do not specify *expression* (or if *expression* is 0), PROGRESS starts a new line only if output is not already positioned at the beginning of a new line.

SPACE(*expression*)

Specifies the number of spaces you want to output. Spaces are not placed between items being PUT unless you use the SPACE option.

EXAMPLE

```
                                                    r-put.p
    DEFINE STREAM s1.

    OUTPUT STREAM s1 TO cus.dat.

    FOR EACH customer:
  ➡   PUT STREAM s1 name "/".
    END.

    OUTPUT STREAM s1 CLOSE.
```

This procedure creates an ASCII file containing the names of each customer. The names are separated from one another by a slash (/). The entire file consists of one very long line.

NOTES

- In the AT, TO, SKIP, and SPACE options, if *expression* is less than or equal to 0, PROGRESS disregards the option.
- The PUT statement never automatically starts a new line. You must use SKIPs to explicitly start new lines.
- The PUT statement uses the default display format for the data type of the field or variable you name in the PUT statement. The PUT statement does not overwrite an area that is already used by a previous format when it displays data. For example:

```
DEFINE VARIABLE myname AS CHARACTER FORMAT "x(8)".
DEFINE VARIABLE mynum AS CHARACTER FORMAT "x(8)".

myname = "abc".
mynum = "123".
OUTPUT TO myfile.
PUT myname AT 8 mynum AT 12.
OUTPUT CLOSE.
```

The output in myfile looks like this:

```
abc
     123
```

because the PUT statement does not overwrite the trailing blanks after "abc".

Use the UNFORMATTED option with the PUT statement to override the format-sensitive display.

- The NULL keyword can be used to output null characters ('\0') in a control sequence. For example, the following two statements write the control sequence ESC 'A' '\0' and 20 NULLs to output stream A.

```
PUT STREAM A CONTROL "~033A" NULL.
PUT STREAM A CONTROL NULL(20).
```

SEE ALSO DEFINE STREAM Statement, DISPLAY Statement, EXPORT Statement, OUTPUT TO Statement, PAGE Statement, PUT SCREEN Statement

PUTBYTE

Replaces a byte in a variable with the integer value of an expression.

SYNTAX

```
PUTBYTE(variable,position) = expression
```

variable
A variable of the datatype raw.

position
An integer value greater than 0 that indicates the byte where PROGRESS places *expression*.

expression
The integer value of a constant, field name, variable name, or any combination of these.

EXAMPLE

rawput.p

```
/*You must connect to a non-PROGRESS demo database to run
  this procedure*/
 DEFINE VAR rl AS RAW.

 FIND customer WHERE cust-num = 26.
 DISPLAY name.
 rl = RAW(name).
 PUTBYTE(rl,1) = ASC('B').
 PUTBYTE(rl,2) = ASC('i').
 PUTBYTE(rl,3) = ASC('l').
 PUTBYTE(rl,4) = ASC('l').

 RAW(name) = rl
 DISPLAY name.
```

This procedure finds the name of customer 26, Jack's Jacks, and stores it in the raw variable rl. The PUTBYTE statement replaces the first four bytes in the name with the specified ASCII values. The procedure then writes the values in rl back into the name field and displays that field. Jack's Jacks becomes Bill's Jacks.

NOTES

- If *position* is greater than the length of *variable*, then PROGRESS makes the new length of *variable* equal to position and pads the gap with nulls.
- If *expression* is less than zero or greater than 255, PROGRESS takes the right most byte of expression to place in *variable*.
- If *variable* is the unknown value, it remains the unknown value.

SEE ALSO GETBYTE, LENGTH Function, LENGTH Statement, RAW Function, RAW Statement

PUT CURSOR Statement

Makes the cursor visible on the screen. In data handling statements such as UPDATE, SET, PROMPT-FOR, and INSERT, PROGRESS handles cursor display so the user knows where the cursor is on the screen. However, if you are getting data from users through the READKEY statement, and that statement is not part of an EDITING phrase, you may want to turn the cursor on so the user can see the location of the cursor while entering data.

SYNTAX

$$\text{PUT CURSOR} \left\{ \begin{array}{l} \text{OFF} \\ \left\{ \begin{array}{l} \text{ROW } \textit{expression} \\ \text{COLUMN } \textit{expression} \end{array} \right\} \ldots \end{array} \right\}$$

OFF

Ends display of the cursor.

ROW *expression*

The row at which you want to display the cursor. In the ROW option, *expression* is a constant, field name, variable name, or any combination of these whose value is an integer that indicates the row at which you want to display the cursor. If you do not use the ROW option, PUT CURSOR displays the cursor at row 1. If you specify a ROW that is outside the screen area, PROGRESS positions the cursor at the screen border closest to the specified row.

COLUMN *expression*

The column at which you want to display the cursor. In the COLUMN option, *expression* is a constant, field name, variable name, or any combination of these whose value is an integer that indicates the column at which you want to display the cursor. If you do not use the COLUMN option, PUT CURSOR displays the cursor at column 1. If you specify a COLUMN that is outside the screen area, PROGRESS positions the cursor at the screen border closest to the specified column.

EXAMPLE

r-cursor.p

```
DEFINE VARIABLE comment AS CHARACTER FORMAT "x(30)"
       EXTENT 4.
DEFINE VARIABLE r       AS INTEGER.
DEFINE VARIABLE c       AS INTEGER.
DEFINE VARIABLE lmargin AS INTEGER INITIAL  5.
DEFINE VARIABLE rmargin AS INTEGER INITIAL 34.
DEFINE VARIABLE ptop    AS INTEGER INITIAL 10.
DEFINE VARIABLE pbot    AS INTEGER INITIAL 13.
DEFINE VARIABLE r-ofst  AS INTEGER INITIAL  9.
DEFINE VARIABLE c-ofst  AS INTEGER INITIAL  4.
FORM SKIP (4) WITH WIDTH 32 ROW 9 COL 4 TITLE "Editor".
MESSAGE "Type text into the editor.  Press F1 to end.".
VIEW.
r = ptop.
c = lmargin.
REPEAT:
  PUT CURSOR ROW r COLUMN c.
  READKEY.
  IF KEYFUNCTION(LASTKEY) = "GO" THEN LEAVE.
  IF KEYFUNCTION(LASTKEY) = "END-ERROR" THEN RETURN.
  IF LASTKEY = KEYCODE("CURSOR-RIGHT") THEN DO:
    c = c + 1.
    IF c > rmargin
    THEN c = lmargin.
    NEXT.
  END.
  IF LASTKEY = KEYCODE("CURSOR-LEFT") THEN DO:
    c = c - 1.
    IF c < lmargin
    THEN c = rmargin.
    NEXT.
  END.
  IF LASTKEY = KEYCODE("CURSOR-DOWN") THEN DO:
    r = r + 1.
    IF r > pbot
    THEN r = ptop.
    NEXT.
  END.
```

(continued)

r-cursor.p - continued

```
    IF LASTKEY = KEYCODE("CURSOR-UP") THEN DO:
      r = r - 1.
      IF r < ptop
      THEN r = pbot.
      NEXT.
    END.
    IF LASTKEY = KEYCODE("RETURN") THEN DO:
      r = r + 1.
      IF r > pbot
      THEN r = ptop.
      c = lmargin.
      NEXT.
    END.
    IF LASTKEY = KEYCODE("BACKSPACE") THEN DO:
      IF c = lmargin AND r = ptop
      THEN NEXT.
      c = c - 1.
      IF c < lmargin
      THEN DO:
      c = rmargin.
      r = r - 1.
      IF r < ptop
      THEN r = ptop.
    END.
    PUT SCREEN ROW r COLUMN c " ".
    OVERLAY(comment[r - r-ofst], c - c-ofst) = " ".
    NEXT.
  END.
  IF LASTKEY >= 32 AND LASTKEY <= 126
  THEN DO:
    PUT SCREEN ROW r COLUMN c KEYLABEL(LASTKEY).
    OVERLAY(comment[r - r-ofst], c - c-ofst) =
           KEYLABEL(LASTKEY).
```

(continued)

```
                                                    r-cursor.p
      c = c + 1.
       IF c > rmargin
       THEN DO:
         c = lmargin.
         r = r + 1.
         IF r > pbot
         THEN r = ptop.
       END.
     END.
   END.

   DISPLAY comment WITH FRAME x NO-LABELS TITLE
           "Comments Array" 1 COLUMN ROW 09 COLUMN 40.
   MESSAGE "Information stored in the comments array.".
```

This procedure uses PUT CURSOR to make the cursor visible on a screen in a sample editor.

When you run the procedure, you see a frame on your screen. You can type text into this editing frame. The procedure reads each key you enter and takes the appropriate action. PUT CURSOR places the cursor at the first row and the first column in the editing frame when you first run the procedure. As you type, the cursor continues to be visible. As the procedure passes through the REPEAT loop for each keystroke, it takes action based on each keystroke and moves the cursor as it takes the action.

The procedure stores the information you type in the comments array, one character at a time. When you finish typing, press F1. The procedure displays the array in which PROGRESS stored the typed information.

NOTES

- You must use the PUT SCREEN statement to display data when you use the PUT CURSOR statement. You also need to define a variable for the cursor position, and increment it as PROGRESS reads the keys entered by the user if you want the cursor to move as the user types.
- The PUT CURSOR statement displays the cursor until you use the PUT CURSOR OFF statement to stop the display.
- Because a cursor is always displayed in an EDITING phrase, using the PUT CURSOR statement in an EDITING phrase (or if you have not issued a PUT CURSOR OFF statement before the phrase) may cause errors.

SEE ALSO PUT SCREEN Statement

PUT SCREEN Statement

Displays a character expression at a specified location on a screen, overlaying any other data that might be displayed at that location.

SYNTAX

```
           ┌ ATTR-SPACE            ┐
           │ COLOR color-phrase    │
PUT SCREEN │ COLUMN expression     │ ...  expression
           │ NO-ATTR-SPACE         │
           └ ROW expression        ┘
```

ATTR-SPACE

There are two ways a terminal can handle screen formatting. It either:

- Reserves a character position on both sides of every field for special screen field attributes, such as underlining or highlighting. These are called "spacetaking" terminals.
- Does not reserve a character position for special field attributes. These are called "nonspacetaking" terminals.

Specifying ATTR-SPACE reserves spaces for field attributes. ATTR-SPACE uses a space on the screen to return the screen to the NORMAL attribute at the location of the PUT SCREEN (even if the screen was already set to NORMAL). See Chapter 7 of the *Programming Handbook* for more information.

COLOR *color-phrase*

The video attributes you want to use to display an expression. When you are displaying data in the first column of a spacetaking terminal, PROGRESS does not display that data with color. If you are displaying data in a column other than column 1, PROGRESS displays the color attribute in the column prior to the current column (current column minus 1). Here is the syntax of *color-phrase:*

```
{ NORMAL
  INPUT
  MESSAGES
  protermcap-attribute
  dos-hex-attribute
  [ BLINK- ][BRIGHT-][fgnd-color ] [ / bgnd-color ]
  [ BLINK- ][ RVV- ][ UNDERLINE- ][ BRIGHT- ] [fgnd-color ]
  VALUE( expression )
}
```

For more information, see the Color Phrase reference page.

COLUMN *expression*

The column at which you want to display an expression. In the COLUMN option, *expression* is a constant, field name, variable name, or any combination of these whose value is an integer that indicates the column at which you want to display an expression. If you do not use the COLUMN option, PUT SCREEN displays the expression at column 1. If you specify a COLUMN that is outside the screen area, PROGRESS disregards the PUT SCREEN statement.

NO-ATTR-SPACE

Does not reserve spaces for field attributes such as underlining and highlighting. See Chapter 7 of the *Programming Handbook* for more information.

If your terminal is a spacetaking terminal, and you display an attribute such as highlighting that requires spaces, you cannot use the NO-ATTR-SPACE option to override that requirement of spaces.

ROW *expression*

The row at which you want to display an expression. In the ROW option, *expression* is a constant, field name, variable name, or any combination of these whose value is an integer that indicates the row at which you want to display an expression. If you do not use the ROW option, PUT SCREEN displays the expression at row 1. If you specify a ROW that is outside the screen area, PROGRESS disregards the PUT SCREEN statement.

expression

A constant, field name, variable name, or any combination of these that results in a character string. The character string can contain control characters and can be as long as you want.

EXAMPLE

The r-putscr.p procedure determines whether a customer's current balance is above or below 0. If it is above 0, they have a credit; if it is below 0, they owe money. The label of the current balance (curr-bal) column is changed based on whether they have a credit or owe money.

If the customer has a credit (curr-bal < 0) the first PUT SCREEN statement displays the value of bal-label (which would be "Customer Credit") in the same color as you see system MESSAGES (usually reverse video).

If the customer owes money (curr-bal > 0) the second PUT SCREEN statement displays the value of bal-label (which would be "Current Balance") in normal display mode.

r-putscr.p

```
DEFINE VARIABLE paid-owed AS DECIMAL.
DEFINE VARIABLE bal-label AS CHARACTER
  FORMAT "x(20)".

FOR EACH customer:
  paid-owed = curr-bal.
  IF paid-owed < 0 /* Customer has a credit */
  THEN DO:
    paid-owed = - paid-owed.
    bal-label = "Customer Credit     ".
  END.
  ELSE
    bal-label = "Unpaid Balance      ".
  DISPLAY cust-num name
          paid-owed LABEL "                    "
          WITH 1 DOWN.
  IF curr-bal < 0
➡ THEN PUT SCREEN COLOR MESSAGES ROW 2 COLUMN 34
    bal-label.
➡ ELSE PUT SCREEN ROW 2 COLUMN 34 bal-label.
END.
```

NOTES

- PROGRESS does not treat values displayed by PUT SCREEN as belonging to frames. That means those expressions can be overwritten by other displays or hides. You should be careful to ensure that values displayed by PUT SCREEN do not overwrite frame fields that are later used for data entry.
- If you use the PUT SCREEN statement in a procedure that runs in batch or background mode, PROGRESS disregards the PUT SCREEN statement.
- The HIDE ALL statement clears the entire screen, including any data displayed by a PUT SCREEN statement.
- The Wyse 75 terminal is spacetaking for some COLOR attributes and nonspacetaking for others. This difference interferes with resetting COLOR MESSAGE (nonspacetaking) back to COLOR NORMAL in a PUT SCREEN statement. If you use WHITE instead of NORMAL whenever you reset color attributes back to normal video attributes, the Wyse 75 will behave like other terminals.
- If you use the PUT SCREEN statement to display data in the message area, the HIDE MESSAGES statement does not necessarily clear that data.

SEE ALSO Color Phrase, DISPLAY Statement, PUT Statement

QUIT Statement

Exits from PROGRESS and returns to the operating system

SYNTAX

```
QUIT
```

EXAMPLE

r-quit.p

```
DEFINE VARIABLE ans AS INTEGER FORMAT "9".
DEFINE VARIABLE proc AS CHARACTER EXTENT 4.

proc[1] = "newcust.p".
proc[2] = "chgcust.p".
proc[3] = "delcust.p".
proc[4] = "prncust.p".
  WITH FRAME threshold NO-BOX NO-LABELS ROW 10
REPEAT:
  FORM
    "     CUSTOMER MAINTENANCE       " SKIP(2)
    "1 -  Create New Customer        " SKIP(1)
    "2 -  Change Existing Customer   " SKIP(1)
    "3 -  Delete Customer            " SKIP(1)
    "4 -  Print Customer List        " SKIP(1)
    "5 -  Exit From PROGRESS         " SKIP(2)
    "SELECTION" ans WITH NO-BOX NO-LABELS
    CENTERED FRAME cusmaint1.
    UPDATE ans WITH FRAME cusmaint1.
➡   IF ans = 5 THEN QUIT.
    ELSE IF ans < 1 OR ans > 4
    THEN DO:
      MESSAGE "Invalid Choice. Try again.".
      UNDO, RETRY.
    END.
    HIDE FRAME cusmaint1.
    RUN VALUE(proc[ans]).
END.
```

This procedure displays a menu. If you choose option 5, Exit From PROGRESS, the procedure processes the QUIT statement.

NOTE

- If QUIT is executed during a transaction, PROGRESS commits the transaction before exiting.

SEE ALSO STOP Statement

R-INDEX Function

Returns an integer that indicates the position of the target string within the source string. In contrast to the INDEX function, the search is performed from right to left.

SYNTAX

```
R-INDEX ( source, target)
```

source

A character expression. This can be a constant, field name, variable name, or any combination of these that results in a character value.

target

A character expression whose position you want to locate in *source*. If *target* does not exist within *source*, R-INDEX returns 0.

The search for the *target* pattern begins at the right-most character of the *source* string. Even though the search is started from the right, the target-position is calculated from the left. For example,

```
R-INDEX("abcd"  , "c")
```

returns a 3 rather than a 2.

EXAMPLE

```
                                                    r-rindex.p
DEFINE VARIABLE rindx AS INTEGER.
DEFINE VARIABLE source AS CHARACTER FORMAT "X(45)".
DEFINE VARIABLE target AS CHARACTER FORMAT "X(45)".
REPEAT:
  PROMPT-FOR source
   LABEL "Enter a character string to do pattern matching:"
        WITH FRAME s1 CENTERED.
  PROMPT-FOR target
   LABEL "Enter a pattern to match in the string:"
        WITH FRAME t1 CENTERED.
➤ rindx = R-INDEX(INPUT source, INPUT target).
  IF rindx <> 0 THEN DO:
    DISPLAY "The target pattern:" INPUT target NO-LABEL
            "last appears in position" rindx NO-LABEL SKIP
      WITH FRAME r1 ROW 12 CENTERED.
     DISPLAY "in the source string:" INPUT source NO-LABEL
      WITH FRAME r1 ROW 12 CENTERED.
  HIDE FRAME r1.
  END.
  IF rindx = 0 THEN DO:
     DISPLAY "The target pattern:" INPUT target NO-LABEL
         "could not be found" SKIP
         WITH FRAME r2 ROW 12 CENTERED.
     DISPLAY "in the source string:" INPUT source NO-LABEL
         WITH FRAME r2 ROW 12 CENTERED.
  HIDE FRAME r2.
  END.
END.
```

This procedure prompts you to enter a character string and a pattern to match against the string. It then displays the starting position of the string where the pattern was found.

NOTES

- If either operand is case-sensitive, then the R-INDEX function is also.
- If either the *source* string or *target* pattern is null, the result is 0.

SEE ALSO INDEX Function, LOOKUP Function

RANDOM Function

Returns a random integer between two integers (inclusive). When a procedure is run from different PROGRESS sessions, the same number may be generated.

SYNTAX

```
RANDOM( low,high )
```

low

An integer expression (a constant, field name, variable name, or any combination of these whose value is integer) that is the lower of the two expressions you are supplying to the RANDOM function.

high

An integer expression that is the higher of the two expressions you are supplying to the RANDOM function.

EXAMPLE

r-random.p

```
DEFINE VARIABLE onum AS INTEGER.
DEFINE VARIABLE olnum AS INTEGER.

DO onum = 1 TO 10 TRANSACTION:
  CREATE order.
  order.order-num = onum.
  order.odate = TODAY.
➡ DO olnum = 1 TO RANDOM(1,9):
    CREATE order-line.
    order-line.line-num = olnum.
    order-line.item-num = olnum.
  END.
END.
```

Often when you set up a database for testing purposes, you want to generate many records without actually keying in data for each record. The r-random.p procedure generates 10 order records and a random number of order-lines for each order record.

RAW Function
(RMS, Rdb, and ORACLE only)

Extracts bytes from a field.

SYNTAX

```
RAW(field [,postition [,length]])
```

field
Any field from which you want to extract bytes.

position
An integer expression (a constant, field name, variable name, or any combination of these that results in an integer value) that indicates the position of the first byte you want to extract from *field*. The default value of *position* is 1.

length
An integer expression that indicates the number of bytes you want to extract from *field*. If you do not use the *length* argument, RAW uses *field* from *position* to end.

EXAMPLE

rawfunc.p

```
/*You must connect to a non-PROGRESS demo database to run
 this procedure*/

DEFINE VAR r1 AS RAW

FIND FIRST cust.
r1 = RAW(name,8,4).
```

This procedure extracts bytes from the name field of the first customer, starting at byte eight, and writes four bytes to the variable r1.

NOTES

- If *position* is less then one, or *length* is less than zero, you receive a run-time error.
- If (*position* + *length* -1) is greater than the length of the field from which you are extracting the bytes, you receive a run-time error.

SEE ALSO GETBYTE, LENGTH, RAW Statement, PUTBYTE

RAW Statement (RMS, Rdb, and ORACLE only)

Writes bytes to a field.

SYNTAX

RAW(*field* [*,position* [*,length*]]) = *expression*

field
The field where you want to store *expression.*

position
An integer expression (a constant, field name, variable name, or any combination of these that results in an integer value) that indicates the position in *field* at which you want to start storing *expression.* The default for *position* is one.

length
An integer expression that indicates the number of positions you want to replace in *field.* If you do not use the *length* argument, RAW puts *expression* into *field* from *position* to end. PROGRESS treats variable length fields and fixed length fields differently — see **NOTES** at the end of this section.

expression
A function or variable name that returns data and results in the bytes that you want to store in *field.*

EXAMPLE

rawdemo4.p

```
/*You must connect to a non-PROGRESS demo database to run
  this procedure*/

DEFINE VAR r1 AS RAW.

FIND FIRST CUST.
DISPLAY name.
r1 = RAW(name).
PUTBYTE (r1,18) = 115.
PUTBYTE (r1,19) = 0.
RAW(name) = r1.
DISPLAY name.
```

This procedure writes the name of the first customer in the database, Second Skin Scuba, to the variable r1. It puts two additional bytes, an ASCII s and a null terminator, on the end of the name, and writes the name back to the database with the RAW statement. The procedure then displays the new name, Second Skin Scubas.

NOTES

- In a variable length field, if (*position* + *length* -1) is greater than the length of *field*, then PROGRESS pads the field with nulls before it performs the replacement.
- In a fixed length field, if (*position* + *length* -1) is greater than the length of *field*, you receive a run-time error. If (*position* + *length* -1) is less then the length of *field*, PROGRESS pads the field with nulls so that it remains the same size.
- On ORACLE and Rdb databasees, if *position*, *length*, or *expression* is equal to the unknown value (?), then *field* becomes unknown.
- If *position* is less than one, or *length* is less than zero, you receive a run-time error.

SEE ALSO GETBYTE, LENGTH Function, LENGTH Statement, RAW Function, PUTBYTE

READKEY Statement

Reads one keystroke from an input source and sets the value of LASTKEY to the keycode of that keystroke. You use the READKEY statement when you want to look at each keystroke a user makes and take some action based on that keystroke.

SYNTAX

```
READKEY [ STREAM stream ] [ PAUSE n ]
```

STREAM *stream*

Specifies the name of a stream. If you do not name a stream, PROGRESS uses the unnamed stream. See the DEFINE STREAM reference page and Chapter 9 of the *Programming Handbook* for more information about streams.

PAUSE *n*

The READKEY statement waits up to *n* seconds for a keystroke. If you do not press a key during that amount of time, READKEY ends, and sets the value in LASTKEY to -1.

PAUSE 0 causes READKEY to immediately return a value. If no character is available, READKEY sets the value of LASTKEY to -1. This form of READKEY is useful for using UNIX pipes or terminal ports to do polling.

EXAMPLE

r-readky.p

```
FOR EACH customer:
  DISPLAY cust-num name address city st
    WITH 1 DOWN.
  MESSAGE
    "To delete this customer, press Y".
  MESSAGE
    "Otherwise, press any other key.".
➡ READKEY.
  IF CHR(LASTKEY) = "Y"
  THEN DELETE customer.
  ELSE
  IF KEYFUNCTION(LASTKEY) = "END-ERROR"
  THEN LEAVE.
END.
```

When the user presses a key, the READKEY statement reads the keystroke and stores the ASCII value of that key (the key code) as the value of LASTKEY. The CHR function converts the ASCII value into a character value. If the character value is a "Y", the customer is deleted. KEYFUNCTION determines the function of the LASTKEY. If that function is END-ERROR, PROGRESS leaves the block, ending the procedure.

NOTES

- If the current input source is a file, then READKEY reads the next character from that file and returns the value of that character (always one of 1 to 255) to LASTKEY. READKEY does not translate periods (.) in the file into the ENDKEY value. It does translate end of line into RETURN (13) but it cannot read any special keys such as function keys.

  When PROGRESS reaches the end of the file, it sets the value of LASTKEY to -2 but does not close the input file. At that point, an APPLY LASTKEY (same as APPLY -2) raises the ENDKEY condition.

- If the current input source is a UNIX pipe, any timer you have set with the PAUSE option may expire before READKEY can read a character. If so, LASTKEY is set to -1.

- If the last key typed is an invalid character sequence, READKEY sets the value of LASTKEY to -1.

- On UNIX System V machines, if input is coming from a pipe, READKEY PAUSE 0 is treated as READKEY PAUSE 1 (i.e. there is always a one second wait for input).

- READKEY counts as an interaction in determining if an UNDO, RETRY should be treated as UNDO, NEXT and whether UNDO, NEXT should be treated as UNDO, LEAVE for purposes of infinite loop protection.

SEE ALSO DEFINE STREAM Statement, LASTKEY Function, Chapter 2 of the *Programming Handbook*

RECID Function

Returns the unique internal identifier of the database record currently associated with the record buffer you name.

SYNTAX

```
RECID( record )
```

record

The name of the record whose RECID you want.

To use the RECID function with a record in a file defined for multiple databases, you must qualify the record's filename with the database name. See the description of the Record Phrase for more information.

EXAMPLE

r-recid.p

```
DEFINE VARIABLE response AS LOGICAL.
DEFINE VARIABLE crecid AS RECID.

REPEAT:
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num NO-LOCK.
  crecid = RECID(customer).
  DISPLAY name.
  response= yes.
  UPDATE response LABEL "Update max-credit?".
  IF response THEN DO:
    FIND customer WHERE RECID(customer) =
         crecid EXCLUSIVE-LOCK.
  UPDATE max-credit.
  END.
END.
```

You may decide that you don't want to lock a record until the user starts to update that record. In this procedure, the FIND statement reads a customer record without locking the record. The RECID function puts the internal database identifier of that record in the crecid variable. If the user decides to update the max-credit field, the procedure re-finds the record using the value in crecid. The second FIND statement reads the record again, this time placing an EXCLUSIVE-LOCK on it. Because the record is first found with NO-LOCK, it is possible for the record to be updated by another user after the first FIND and before the second.

NOTES

- The RECID function is useful when you want to rapidly retrieve a previously identified record, even if that record has no unique index.
- If you want a called procedure to use the same record as a calling procedure, you can use the RECID function to ensure that you are retrieving the identical record. A SHARED variable is used to communicate the recid of a record from one procedure to another. The second procedure can then find the same record. This is an alternative to using shared buffers.
- Avoid storing recid values in database fields because those recids will change if you dump and reload the database.
- You do not need to explicitly check to see if a record is AVAILABLE before using the RECID function. The RECID function returns the unknown value (?) if a record cannot be accessed.

  The first example shows the assignment of a RECID only when a record can be accessed:

```
DISPLAY
   (IF AVAILABLE customer
   THEN RECID(customer) ELSE ?).
```

  It is sufficient to directly assign a RECID even if a record cannot be found, as shown in the next example.

```
FOR EACH customer:
   DISPLAY cust-num.
END.

DISPLAY RECID(customer).
```

SEE ALSO DEFINE BUFFER Statement, DEFINE VARIABLE Statement

Record Phrase

Identifies the record you want to retrieve with a FIND statement, the set of records to retrieve using a FOR EACH statement, or the constraints on the records being preselected in a DO or REPEAT block.

The Record Phrase syntax describes three kinds of information:

1. Qualification of the record or records to access in the file.
2. Which index to use when locating records.
3. The type of record lock to apply when the records are read.

SYNTAX

```
record  [ constant  ]  [ WHERE [expression]                                       ] ...
                       [ USING [ FRAME frame ]  field [ AND field  ] ...          ]
                       [ OF file                                                  ]
                       [ USE-INDEX index                                          ]

                       [ SHARE-LOCK      ]
                       [ EXCLUSIVE-LOCK  ]
                       [ NO-LOCK         ]
```

record

The name of a file or of a buffer you named in a DEFINE BUFFER statement.

To access a record in a file defined for multiple databases, you must qualify the record's filename with the database name. Use the following syntax to refer to a record in a file for a specific database:

```
dbname.filename
```

You do not need to qualify the reference if *record* is the name of a buffer.

constant

The value of a single component, unique, primary index for the record you want. For example:

```
FIND customer 1.
```

PROGRESS converts this FIND statement with the *constant* option of 1 to the following:

```
FIND customer WHERE cust-num = 1.
```

The cust-num field is the only component of the primary index of the customer file.

If you use the *constant* option, you must use it just once in a single Record Phrase and it must precede any other options in the Record Phrase.

WHERE *expression*

Qualifies the records you want to access. The *expression* is a constant, field name, variable name, or any combination of these whose value you want to use to select records. You may use the WHERE keyword even if you do not supply an *expression*. For example:

```
FOR EACH customer WHERE {*}
```

USING *field* [AND *field*]...

One or more names of indexed fields for selecting records. The field you name in this option must have been entered previously, usually with a PROMPT-FOR statement.

The USING option translates into an equivalent WHERE option. For example:

```
PROMPT-FOR customer.cust-num.
FIND customer USING cust-num.
```

This FIND statement is the same as the following:

```
FIND customer WHERE customer.cust-num =
        INPUT customer.cust-num.
```

Cust-num is a nonabbreviated index. However, look at this example:

```
PROMPT-FOR customer.name.
FIND customer USING cust-name.
```

If the name field is an abbreviated index of the customer file, PROGRESS converts the FIND statement with the USING option into the following statement:

```
FIND customer WHERE customer.name
         BEGINS INPUT name.
```

Note that field can be expanded to be [FRAME *frame*] *field*.

field
Where field is [INPUT] [FRAME *frame*] [[*dbname*.] *file*.] *field* [*subscript*].

subscript
Where subscript is [*expression* [FOR *n*]]. Note that the outermost brackets here are required.

OF *file*
Qualifies the records by relating the record to a record in another file. For example:

```
PROMPT-FOR order.order-num.
FIND order USING order-num.
DISPLAY order.
FIND customer OF order.
DISPLAY customer.
```

In this example, the OF option relates the order file to the customer file, telling PROGRESS to select the customer record related to the order record currently being used. When you use OF, all fields participate in match criteria, if an index is multi-field. The relationship is based on having a UNIQUE index in one file. PROGRESS converts the FIND statement with the OF option to the following:

```
FIND customer WHERE customer.cust-num =
        order.cust-num.
```

You can always access related files using WHERE whether or not the field names of the field or fields that relate the files have the same name.

SHARE-LOCK

Tells PROGRESS to put a SHARE-LOCK on records as they are read. Other users can still read a record that is share locked but they cannot update it. By default, PROGRESS puts a SHARE-LOCK on a record when it is read, and automatically puts an EXCLUSIVE-LOCK on a record when it is modified (unless the record is already EXCLUSIVE-LOCKed).

If you use the SHARE-LOCK option and PROGRESS tries to read a record that is EXCLUSIVE-LOCKed by another user, PROGRESS waits to read the record until the EXCLUSIVE-LOCK is released. PROGRESS displays a message to the user of that procedure, identifying the file that is in use, the userid of the user and the tty of the terminal using the file.

If you are using a record from a work file, PROGRESS disregards the SHARE-LOCK option.

EXCLUSIVE-LOCK

Tells PROGRESS to put an EXCLUSIVE-LOCK on records as they are read. Other users cannot read or update a record that is EXCLUSIVE-LOCKed, except by using the NO-LOCK option. They can access that record only when the EXCLUSIVE-LOCK is released. PROGRESS automatically puts a SHARE-LOCK on a record when it is read and automatically puts an EXCLUSIVE-LOCK on a record when it is updated.

If a record is read specifying EXCLUSIVE-LOCK, or if a lock is being automatically changed to EXCLUSIVE-LOCK by an update, another user's read or update will wait if any other user has the record SHARE-LOCKed or EXCLUSIVE-LOCKed.

When a procedure tries to use a record that is EXCLUSIVE-LOCKed by another user, PROGRESS displays a message identifying the file that is in use, the userid of the user and the tty of the terminal using the file.

If you are using a record from a work file, PROGRESS disregards the EXCLUSIVE-LOCK option.

USE-INDEX *index*

Identifies the index you want to use while selecting records. If you do not use this option, PROGRESS selects an index to use based on the criteria specified with the WHERE, USING, OF, or *constant* options.

NO-LOCK

Tells PROGRESS to put no locks on records as they are read and to read a record even if another user has it EXCLUSIVE-LOCKed. Other users can read and update a record that is not locked. By default, PROGRESS puts a SHARE-LOCK on a record when it is read, and automatically puts an EXCLUSIVE-LOCK on a record when it is updated (unless the record is already EXCLUSIVE-LOCKed). A record that has been read NO-LOCK must be re-read before it can be updated. For example:

```
DEFINE VARIABLE rid AS RECID.
rid = RECID(customer).
FIND customer WHERE
  RECID(customer) = rid EXCLUSIVE-LOCK.
```

If a procedure finds a record and it places it in a buffer using NO-LOCK and you then re-find that record using NO-LOCK, PROGRESS does not reread the record. Instead, it uses the copy of the record that is already stored in the buffer.

When you read records with NO-LOCK, you have no guarantee as to the overall consistency of those records because another user may be in the process of changing them. For example, when values are assigned to indexed fields for a newly created record or are modified in an existing record, the index is immediately updated to reflect the change, but the copy of the data record in the buffers used by the database server may not be updated until later in the transaction. For example, the procedure:

```
FOR EACH customer WHERE cust-num >
100 NO-LOCK:
  DISPLAY cust-num.
END.
```

may display a cust-num of 0 if another user's active transaction has created a record and assigned a value to the indexed field cust-num that is greater than 100.

If you are using a record from a work file, PROGRESS disregards the NO-LOCK option.

EXAMPLES

```
                                                  r-recph.p
➡ FOR EACH customer WHERE max-credit GE 100,
    EACH order OF customer:
      DISPLAY customer.cust-num customer.name
              max-credit order.order-num odate
              order.terms.
  END.
```

In the r-recph.p procedure, there are two Record phrases:

1. customer WHERE max-credit GE 100
2. order OF customer

Using these Record phrases, the FOR EACH block reads a customer record only if it has a max-credit value greater than 100 and at least one order record associated with it.

```
                                                  r-recph2.p
  REPEAT:
➡   FIND NEXT customer USE-INDEX zip WHERE
      name BEGINS "S" EXCLUSIVE-LOCK.
    UPDATE name zip phone.
  END.
```

In the r-recph2.p procedure, there is one Record phrase:

```
  customer USE-INDEX zip WHERE
           name BEGINS "S" EXCLUSIVE-LOCK
```

Using the index named zip rather than the cust-num index (the primary index for the customer file), the FIND statement reads only those customer records having a name that begins with an “s”. The FIND also places an EXCLUSIVE-LOCK on each record as it is read. This lock is released at the end of the REPEAT block.

In the output of this procedure, all the customer names begin with “s” and the customers are displayed in order by zip code.

SEE ALSO DO Statement, FIND Statement, FOR Statement, REPEAT Statement

RELEASE Statement

Verifies that a record complies with mandatory field and unique index definitions, and clears the record from the buffer, writing it back to the database if it has been changed. You use the RELEASE statement for share-locked records only.

SYNTAX

```
RELEASE record
```

record

The name of a record buffer.

To use RELEASE with a record in a file defined for multiple databases, you must qualify the record's filename with the database name. See the description of the Record Phrase for more information.

EXAMPLE

```
                                                r-releas.p
   DEFINE VARIABLE prt LIKE syscontrol.printr.

   FIND FIRST syscontrol.
   prt = syscontrol.printr.
➡  RELEASE syscontrol.
   DISPLAY prt.
   IF prt THEN OUTPUT TO PRINTER.
```

The syscontrol file in the demo database is a file that is used to store different kinds of application system information. There is just a single record in the syscontrol file. That record has a field for each kind of system information. One of those fields is printr. The printr field is a logical field and contains either a YES or a NO value. This value determines whether or not reports generated by the application are to be sent to the printer.

The r-releas.p procedure reads a record from the syscontrol file, assigns to the prt variable the value of the syscontrol.printr field, and releases the syscontrol record. This is just a partial procedure. After the OUTPUT TO statement, the procedure could go on to produce a report or to call one or more subroutines that produce reports.

When the FIND statement in the r-releas.p procedure reads the syscontrol record into a record buffer, PROGRESS automatically puts a SHARE-LOCK on that record. As long as the record is SHARE-LOCKed, other users can look at the record, but no other users can update the record. That SHARE-LOCK is held until the end of the scope of the syscontrol record. At the end of the record scope, the record buffer is cleared.

The scope of a record is the outermost block in which that record is referenced. The outermost block in which the syscontrol record is referenced in this example is the procedure block. That means that the SHARE-LOCK is held until the procedure ends. During that time, if any other user tries to update any of the fields in the syscontrol file, they will be told to wait.

Suppose the r-releas.p procedure produces a very long report or calls subroutines that produce a series of reports. The r-releas.p procedure could take quite a while to complete. Therefore, it is not a good idea to keep the record SHARE-LOCKed for the duration of the r-releas.p procedure.

The RELEASE statement tells PROGRESS to clear the record buffer. At that time, the scope of the record ends and the SHARE-LOCK is released, making the syscontrol record available for updates by other users.

NOTE

- An ERROR occurs if the validation of the record fails. This can happen only with newly created records.

SEE ALSO Chapter 8 of the *Programming Handbook*

REPEAT Statement

Begins a block of statements that are processed repeatedly until the block ends in one of several ways.

BLOCK PROPERTIES

Iteration, Record Scoping, Frame Scoping, Transactions by default.

SYNTAX

```
[ label : ] REPEAT
   [ FOR record [ , record ] ... ]
   [ PRESELECT[ EACH ] record-phrase
            [ , [ EACH ] record-phrase ] ...
            [ [ BREAK ] { BY expression [ DESCENDING ]} ... ] ]
   [ variable = expression1   TO expression2   [ BY k ]]
   [ WHILE expression   ]
   [ TRANSACTION ]
   [ ON ENDKEY-phrase]
   [ ON ERROR-phrase ]
   [ frame-phrase   ]
```

FOR *record* [, *record*]

Names a record buffer and scopes the buffer to the block. The scope of a record determines when the buffer is cleared and the record is written back to the database. See Chapter 5 of the *Programming Handbook* for more information on record scoping.

To access a record in a file defined for multiple databases, you must qualify the record's filename with the database name. See the description of the Record Phrase for more information.

PRESELECT [EACH] *record-phrase*

Goes through a file to select the records that meet the criteria you specify in a *record-phrase*. PRESELECT creates a temporary index containing pointers to each of the preselected records in the database file. You can then use other statements, such as FIND NEXT, to process those records.

If you name multiple files with PRESELECT, any sorting you specify applies to all files. If you then do a FIND on the last file in the PRESELECT list, records are read into the buffers for all of the files in the list.

To process a multi-file collection gathered by the PRESELECT option, use the last file named in the collection when you want to read the selected records. PROGRESS will automatically retrieve records from the other files.

For example:

```
REPEAT PRESELECT EACH order, customer OF order,
  EACH order-line OF order BY order.odate BY cust.name
  BY order-line.item-num:

  FIND NEXT order-line.
END.
```

The PRESELECT option in this example selects the logically "joined" record consisting of order, order-line, and customer, and makes all of these records available in the REPEAT block.

The *record-phrase* option identifies the criteria to use when preselecting records. Here is the syntax for *record-phrase*:

```
record [ constant ] ┌ WHERE expression                ┐ ...
                    │ USING field [ AND field ]...    │
                    │ OF file                         │
                    └ USE-INDEX index                 ┘

                    ┌ SHARE-LOCK     ┐
                    │ EXCLUSIVE-LOCK │
                    └ NO-LOCK        ┘
```

For more information about *record-phrase*, see the Record Phrase reference page.

BREAK
: Indicates that subgroups will be used for the purposes of aggregation and use of the FIRST, LAST, FIRST-OF and LAST-OF functions. If you use BREAK you must also use BY.

BY *expression* [DESCENDING]
: Sorts the preselected records by the value of *expression*. If you do not use the BY option, PRESELECT sorts the records in order by the index used to extract the records. The DESCENDING keyword sorts the records in descending order as opposed to the default ascending order.

variable = *expression1* TO *expression2* [BY *k*]

Indicates the name of a field or variable whose value you are incrementing in a loop. *expression1* is the starting value for *variable* on the first iteration of the loop. *k* is the amount to add to *variable* after each iteration and must be a constant. When *variable* exceeds *expression2* (or is less than *expression2* if *k* is negative) the loop ends. Because *expression1* is compared to *expression2* at the start of the first iteration of the block, the block may be executed 0 times. *expression2* is reevaluated on each iteration of the block.

WHILE *expression*

Indicates the condition under which you want the REPEAT block to continue processing the statements within it. The block iterates as long as the condition specified by the expression is true. The expression is any combination of constants, field names, and variable names that yields a logical value.

TRANSACTION

Identifies the REPEAT block as a system transaction block. PROGRESS starts a system transaction for each iteration of a transaction block if there is not already an active system transaction. See Chapters 5 and 8 of the *Programming Handbook* for more information about transactions.

ON ENDKEY-*phrase*

Describes the processing that takes place when the ENDKEY condition occurs during a block. Here is the syntax for the ON ENDKEY-*phrase:*

```
                                  , LEAVE [ label2 ]
ON ENDKEY  UNDO [ label1 ]      [ , NEXT  [ label2 ] ]
                                  , RETRY [ label2 ]
                                  , RETURN
```

For more information about ON ENDKEY-*phrase*, see the ON ENDKEY Phrase reference page.

ON ERROR-*phrase*

Describes the processing that takes place when there is an error during a block.

```
                                  , LEAVE [ label2 ]
ON ERROR  UNDO [ label1 ]       [ , NEXT  [ label2 ] ]
                                  , RETRY [ label2 ]
                                  , RETURN
```

For more information, see the ON ERROR Phrase reference page.

frame-phrase

Specifies the overall layout and processing properties of a frame. Here is the syntax of *frame-phrase*. For more information, see the Frame Phrase reference page.

```
       [ ACCUM                                                  ]
       [ ATTR-SPACE                                             ]
       [ CENTERED                                               ]
       [         { [ DISPLAY ] color-phrase }                   ]
       [ COLOR   { PROMPT color-phrase      }  ...              ]
       [ COLUMN  expression                                     ]
       [ n COLUMNS                                              ]
       [ DOWN                                                   ]
       [ expression    DOWN                                     ]
       [ FRAME frame                                            ]
WITH   [ NO-ATTR-SPACE                                          ]   ...
       [ NO-BOX                                                 ]
       [ NO-HIDE                                                ]
       [ NO-LABELS                                              ]
       [ NO-UNDERLINE                                           ]
       [ NO-VALIDATE                                            ]
       [ OVERLAY                                                ]
       [ PAGE-BOTTOM                                            ]
       [ PAGE-TOP                                               ]
       [ RETAIN n                                               ]
       [ ROW  expression                                        ]
       [ SCROLL  n                                              ]
       [ SIDE-LABELS                                            ]
       [ TITLE [ COLOR  color-phrase      ] expression          ]
       [ TOP-ONLY                                               ]
       [ WIDTH  n                                               ]
```

EXAMPLE

r-rpt.p

```
DEFINE VARIABLE selection AS INTEGER FORMAT "9".

FORM SKIP(3)
  "0 - Exit" AT 32
  "1 - Edit Customer File" AT 32
  "2 - List Customer File" AT 32
  "3 - Edit Item File" AT 32
  "4 - List Item File" AT 32
  "Enter Choice" TO 30 selection AUTO-RETURN
  HEADER "Application Name" "Master Menu" AT 34
  "Company" TO 79 WITH NO-BOX NO-LABELS
  CENTERED FRAME menu.

REPEAT ON ENDKEY UNDO, RETRY:
  UPDATE selection WITH FRAME menu.
  HIDE FRAME menu.
  IF selection = 0 THEN LEAVE.
  ELSE IF selection = 1 THEN RUN custedit.p.
  ELSE IF selection = 2 THEN RUN custrpt.p.
  ELSE IF selection = 3 THEN RUN itemedit.p.
  ELSE IF selection = 4 THEN RUN itemrpt.p.
  ELSE DO:
    BELL.
    MESSAGE "Not a valid choice, try again".
  END.
END.
```

In this menu procedure, if you press END-ERROR (F4) or ENDKEY while the procedure is prompting you for your menu selection, any data you have entered as a selection is undone and the procedure continues to prompt you for a menu selection.

NOTE

- Within a REPEAT block, if you are using the FIND NEXT or FIND PREV statement and you change the value of an index field, PROGRESS makes that change in the index table at the end of the UPDATE or SET statement. Therefore, if you change the value so that the record comes later in the index table, you will see the record again if you are doing FIND NEXTs. If you change the value so that the record comes earlier in the index table, you will see the record again if you are doing FIND PREVs. For example:

```
REPEAT:
  FIND NEXT customer.
  UPDATE cust-num.
END.
```

In this example, if you change customer 1 to customer 300, you will see that customer record again at the end of the procedure.

When you use the PRESELECT option, PROGRESS builds a special index table that does not get updated when index values change. For example, add the PRESELECT option to the above example:

```
REPEAT PRESELECT EACH customer:
  FIND NEXT customer.
  UPDATE cust-num.
END.
```

In this example, if you change customer 2 to customer 200, you will not see that customer record until you look it up with a new procedure.

SEE ALSO DO Statement, Frame Phrase, ON ENDKEY Phrase, ON-ERROR Phrase

RETRY Function

Returns a TRUE value if the current block is being reprocessed after a previous UNDO, RETRY.

SYNTAX

```
RETRY
```

EXAMPLE

r-retry.p

```
REPEAT:
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num.
  IF NOT RETRY
  THEN DISPLAY name address city st zip.
  SET name address city st zip.
  IF zip = 0 THEN UNDO, RETRY.
END.
```

This procedure bypasses the display of the customer data when the REPEAT block is being retried (if customer data is changed and the zip code is entered as 0). When you run this procedure, notice that even though the procedure has undone any data that you entered (in the event that you entered a zip of 0), the data still appears on the screen. That data is saved in the screen buffers but is not stored in the customer record buffer. If you do not use the RETRY function, the DISPLAY statement would be reprocessed and the previous values for the customer fields would be displayed, overwriting the data that was entered in error.

NOTE

- The use of the RETRY function in a block turns of the default error processing, resulting in no infinite loop protection for the block.

SEE ALSO UNDO Statement, Chapter 8 of the *Programming Handbook*

RETURN Statement

Leaves the procedure block, returning to the calling procedure, or if there was no calling procedure, to the PROGRESS editor.

SYNTAX

```
RETURN
```

EXAMPLES

r-return.p

```
DEFINE NEW SHARED VARIABLE nfact AS INTEGER
  LABEL "N Factorial".
DEFINE VARIABLE n AS INTEGER FORMAT "9" LABEL "N".

REPEAT:
  SET n SPACE(5).
  nfact = n.
  RUN r-fact.p.
  DISPLAY nfact.
END.
```

r-fact.p

```
   DEFINE SHARED VARIABLE nfact AS INTEGER.
   DEFINE VARIABLE i AS INTEGER.

   i = nfact.
   nfact = nfact - 1.
   IF nfact <= 1 THEN DO:
     nfact = i.
➤    RETURN.
   END.

   RUN r-fact.p.

   nfact = nfact * i.
```

The r-fact.p procedure calculates the factorial of a number entered in procedure r-return.p. The factorial of a number is the result of multiplying together all of the integers less than or equal to that number (e.g. 3 factorial is 3 * 2 * 1 = 6). The r-fact.p procedure is called recursively because (*n* factorial) is *n* * ((*n* - 1) factorial). Note that this is not the most efficient way to calculate factorials, but in other applications such as bill of material explosions, recursive procedures are a very effective technique.

REVOKE Statement (SQL)

Allows the owner or any user who holds the GRANT OPTION on a table or view to revoke privileges on that table or view.

SYNTAX

```
REVOKE
  { ALL [ PRIVILEGES ] |
   { SELECT |
     INSERT |
    DELETE |
    { UPDATE [(column-list)] }        [,...] } }
  ON table-name FROM {grantee-list | PUBLIC}
```

ALL [PRIVILEGES]
: Revokes all privileges that the revoking user has (SELECT, INSERT, DELETE, and UPDATE) from the specified users. This only revokes the privileges that the user has. If the user revoking the privileges is lacking one or more of these, they are not revoked from others.

SELECT
: Revokes the SELECT privilege from the specified users.

INSERT
: Revokes the INSERT privilege from the specified users.

DELETE
: Revokes the DELETE privilege from the specified users.

UPDATE [(*column-list*)]
: Revokes the UPDATE privilege from the specified users. You can list the columns on which you want to revoke the UPDATE privilege. If you specify the keyword UPDATE but omit the column list, the UPDATE privilege is revoked on all columns of the specified table.

ON *table-name*
: The name of the table or view on which you want to revoke privileges.

FROM *grantee-list*

The users from whom you want to revoke privileges. You can specify either a list of user names or the keyword PUBLIC which revokes the privileges from all users.

EXAMPLES

```
REVOKE SELECT
    ON doc
    FROM PUBLIC.
```

```
REVOKE INSERT, UPDATE
    ON employee
    FROM kathy.
```

NOTES

- The REVOKE statement can be used only in interactive SQL.
- You cannot revoke privileges from the owner of a table or view.

ROLLBACK WORK Statement (SQL)

Discards all database changes affected by SQL data manipulation statements since the previous COMMIT WORK or ROLLBACK WORK statement or since the beginning of the session.

SYNTAX

```
ROLLBACK WORK
```

EXAMPLES

```
ROLLBACK WORK.
```

NOTES

- The ROLLBACK WORK statement is disabled by default in interactive SQL. To enable it, enter the command COMMIT OFF in the PROGRESS editor. **You cannot use this statement in a PROGRESS procedure.** Associated commands are COMMIT ON to disable the COMMIT WORK statement and COMMIT STATUS to display the COMMIT ON/OFF status.

SEE ALSO COMMIT ON, COMMIT OFF, COMMIT STATUS, and COMMIT WORK statements.

ROUND Function

Rounds a decimal expression to a specified number of places after the decimal point.

SYNTAX

```
ROUND( expression,precision )
```

expression
A decimal expression (a constant, field name, variable name, or any combination of these that results in a decimal value).

precision
A non-negative integer expression whose value is the number of places you want in the decimal result of the ROUND function.

EXAMPLE

r-round.p

```
FOR EACH customer:
  DISPLAY cust-num name max-credit.
  max-credit =
    ROUND( (max-credit * 1.1) / 100, 0) * 100.
  PAUSE.
  DISPLAY max-credit.
END.
```

This procedure increases all max-credit values by 10 percent, rounding those values to the nearest hundred dollars.

SEE ALSO TRUNCATE Function.

RUN Statement

Runs (calls) a PROGRESS procedure from within a procedure. When the called procedure is completed, the calling procedure resumes at the statement after the RUN statement.

SYNTAX

```
RUN { procedure             } [ ( parameter [ , parameter ] ) ] [ argument ] ...
    { VALUE( expression )   }
```

procedure
The name of the procedure you want run. Under UNIX, procedure names are case-sensitive; under DOS, OS/2, BTOS/CTOS, and VMS they are not. If the procedure you name has an unqualified path name, PROGRESS searches the directories and libraries listed on the PROPATH environment variable, in addition to the PROGRESS system directory, and the prodemo and proguide subdirectories.

VALUE*(expression)*
An expression (a constant, field name, variable name, or any combination of these) whose value is the name of a procedure you want to run.

parameter
A runtime parameter to be passed to the called procedure. A *parameter* has the following syntax.

```
{  [ INPUT ]  expression                  }
{  OUTPUT { field    }                    }
{         { variable }                    }
{  INPUT-OUTPUT { field    }              }
{               { variable }              }
```

A *runtime parameter list* is a parenthesized list of comma-separated parameters and must precede any *arguments*. An *expression* is a constant, field name, variable name, or any combination of these. INPUT is assumed if you do not supply a keyword. OUTPUT and INPUT-OUTPUT parameters must be either record fields or program variables.

Parameters must be defined in the called procedure (see the DEFINE PARAMETER statement). They must be passed in the same order as they are defined, and they must have the same data types.

argument

A constant, field name, variable name, or any combination of these that results in an argument that you want to pass to the procedure you are running.

When you pass arguments to a procedure, PROGRESS converts those arguments to character format. PROGRESS recompiles the called procedure, substituting arguments, and then runs the procedure. You cannot precompile a procedure to which you pass arguments. (If you use shared variables instead of arguments, the procedure can be precompiled. This makes for more efficient code.)

EXAMPLE

r-run.p

```
DEFINE VARIABLE selection AS CHARACTER LABEL
  "Enter Program Choice" FORMAT "x(1)".
DEFINE VARIABLE programs AS CHARACTER
  FORMAT "X(15)" EXTENT 5.

programs[1] = "custrpt.p".
programs[2] = "custedit.p".
programs[3] = "ordrpt.p".
programs[4] = "ordedit.p".
programs[5] = "r-exit.p".

REPEAT:
  FORM HEADER TODAY "MASTER MENU" AT 35
    STRING(TIME,"hh:mm") TO 79.
  FORM SKIP(3)
       "1 - Customer Listing" AT 30
       "2 - Customer Update" AT 30
       "3 - Order Listing" AT 30
       "4 - Order Update" AT 30
       "5 - Quit System" AT 30
       selection COLON 28 AUTO-RETURN
         WITH SIDE-LABELS NO-BOX 1 DOWN.

  UPDATE selection
    VALIDATE(INDEX("12345",selection) NE 0,
             "Not a valid choice").
  HIDE ALL.
➡ RUN VALUE(programs[INDEX("12345",selection)]).
END.
```

This procedure displays a simple menu. The user's selection is stored in the selection variable. The INDEX function returns an integer value indicating the position of the user's selection in a string of characters ("12345"). If the value in the selection variable is not in the list of values, the INDEX function returns a 0. The VALIDATE statement ensures that the INDEX function did not return 0. If it did return 0, VALIDATE displays the message "not a valid choice."

In the RUN statement, the INDEX function returns the position of the user's selection in a character string. Suppose you chose option 2 from the menu. That option occupies the second position in the character string "12345." Therefore, the INDEX function returns the number 2. Using this number, the RUN statement reads, RUN VALUE(programs[2]). According to the assignments at the top of the procedure, the value of programs[2] is custedit.p. Now the RUN statement reads, RUN custedit.p, and the r-run.p procedure runs the custedit.p procedure.

NOTES:

- Procedures can be run recursively (a procedure can run itself).
- In PROGRESS Version 4, you receive an error when you try to run a procedure whose object file is out of date with the database time stamp. This happens if you delete file, field, or index definitions in the Dictionary after compiling and saving the procedure.
- In PROGRESS Version 5, you receive an error when you try to run a procedure whose object file time stamp does not match the time stamp of any database file it references. The procedure must be recompiled and saved. The following actions change a database file's time stamp, invalidating any precompiled procedures that reference the file:
  - Deleting the file.
  - Adding or deleting a field definition in the file.
  - Adding or deleting an index definition in the file.
- The called procedure uses any arguments passed to it by the calling procedure by referring to those arguments as numbers enclosed in braces. The first argument is {1}, the next is {2}, and so on. Any arguments the called procedure does not use are ignored, and any missing arguments are treated as null values. (Note that the null value is a legal WHERE or WITH clause, but its occurrence can cause an error at other points in a called procedure.)

- When you run a procedure, PROGRESS loads that procedure into the edit buffer. If the main procedure calls a subprocedure, PROGRESS loads the subprocedure into the edit buffer right next to the main procedure. PROGRESS continues this scheme each time you run another subprocedure, as long as there is room in the edit buffer. This method of loading procedures means that PROGRESS does not need to load a procedure each time it is run.

  If you are running an application that uses many subprocedures, you may want to increase the size of the edit buffer. See Chapter 3 of *System Administration II: General* for more information on how to use the start-up option to increase the edit buffer size.

- When you run a procedure, PROGRESS checks all the directories and libraries in PROPATH to look for a useable object file of the same name and to check if the procedure was modified since the last time it was run. If there is a usable object file, there is no point in performing the compilation. The RUN statement always uses an existing object file before using a session compile version of a procedure.

  If you do not want PROGRESS to do this checking, use the Quick Request (-q) option. See Chapter 3 of *System Administration II: General* for more information on this option.

SEE ALSO { } Include File, { } Argument Reference, COMPILE Statement, DEFINE PARAMETER Statement.

SCREEN-LINES Function

Returns the number of screen lines you can use to display frames. This value is three less than the total display lines available on the screen.

SYNTAX

```
SCREEN-LINES
```

EXAMPLE

```
r-scrnln.p

DEFINE VARIABLE nbrdown AS INTEGER.

IF SCREEN-LINES > 21
THEN nbrdown = 7.
ELSE nbrdown = 6.

FOR EACH customer WITH nbrdown DOWN:
  DISPLAY cust-num name address city st.
END.
```

Here, a different number of customer records is displayed depending on the number returned by the SCREEN-LINES function.

SCROLL Statement

Moves data up or down in a frame with multiple rows. Use the SCROLL statement to scroll data up or down when you add or delete a line in a frame (often a scrolling frame).

SYNTAX

```
                               ┌ UP   ┐
SCROLL [ FROM-CURRENT ]        └ DOWN ┘ [ frame-phrase ] ...
```

FROM-CURRENT

Scrolls rows of data at or below the current cursor location UP or DOWN. When scrolling UP, a new line opens at the bottom of the frame. When scrolling DOWN, a new line opens at the current cursor location. For example, the Original Frame below shows four rows of data. The highlighted bar is the currrent cursor position and the frame is a scrolling frame. On the right, the SCROLL FROM-CURRENT Statement moves item 00002 up and off the screen, and opens a line at the bottom of the frame.

Original Frame

Item-num
00001
00002
00003
00004

After SCROLL FROM-CURRENT Statement

Item-num
00001
00003
00004

If you do not use the FROM-CURRENT option, then the entire frame scrolls up or down and the newly opened line appears at the top or bottom of a frame, respectively.

FROM-CURRENT limits scrolling to the range from the current cursor position to the bottom of the frame.

UP

Scrolls rows of data up and off the frame and opens a line at the bottom of the frame. UP is the default. For example, the Original Frame below shows four rows of data. The highlighted bar is the current cursor position and the frame is a scrolling frame. On the right, the SCROLL Statement moves item 00001 up and off the screen, and opens a line at the bottom of the frame.

Original Frame

Item-num
00001
00002
00003
00004

After SCROLL Statement

Item-num
00002
00003
00004

DOWN

Scrolls rows of data down and off the frame and opens a line at the top of the frame. For example, the Original Frame below shows four rows of data. The highlighted bar is the current cursor position and the frame is a scrolling frame. On the right, the SCROLL FROM-CURRENT DOWN statement opens a line in the frame at the current cursor location and moves the other rows down and off the frame.

Original Frame

Item-num
00001
00002
00003
00004

After SCROLL FROM-CURRENT DOWN Statement

Item-num
00001
▨
00002
00003

In the next example, the SCROLL DOWN statement opens a line at the top of the frame and moves the other rows of data down and off the frame.

Original Frame

Item-num
00001
00002
00003
00004

After SCROLL DOWN Statement

Item-num
00001
00002
00003

frame-phrase

Specifies the overall layout and processing properties of a frame. Here is the syntax of *frame-phrase*:

```
       [ ACCUM                                                ]
       [ ATTR-SPACE                                           ]
       [ CENTERED                                             ]
       [        { [ DISPLAY ] color-phrase  }                 ]
       [ COLOR  { PROMPT color-phrase       }  ...            ]
       [ COLUMN  expression                                   ]
       [ n COLUMNS                                            ]
       [ DOWN                                                 ]
       [ expression      DOWN                                 ]
       [ FRAME frame                                          ]
       [ NO-ATTR-SPACE                                        ]
 WITH  [ NO-BOX                                               ]
       [ NO-HIDE                                              ]  ...
       [ NO-LABELS                                            ]
       [ NO-UNDERLINE                                         ]
       [ NO-VALIDATE                                          ]
       [ OVERLAY                                              ]
       [ PAGE-BOTTOM                                          ]
       [ PAGE-TOP                                             ]
       [ RETAIN n                                             ]
       [ ROW  expression                                      ]
       [ SCROLL  n                                            ]
       [ SIDE-LABELS                                          ]
       [ TITLE [ COLOR   color-phrase      ] expression       ]
       [ TOP-ONLY                                             ]
       [ WIDTH n                                              ]
```

For more information on *frame-phrase*, see the Frame Phrase reference page.

EXAMPLES

r-scroll.p

```
DEFINE VARIABLE ans AS CHARACTER FORMAT "x".

FORM cust.cust-num cust.name max-credit WITH FRAME cust
     CENTERED 10 DOWN.

FORM  "1 - scroll up" SKIP
      "2 - scroll from-current up" SKIP
      "3 - scroll down" SKIP
      "4 - scroll from-current down" SKIP
      "5 - scroll from-current"
WITH FRAME instruct CENTERED TITLE "Instructions:".

VIEW FRAME cust.

REPEAT WHILE FRAME-LINE(cust) <= FRAME-DOWN(cust):
  FIND NEXT customer.
  DISPLAY cust-num name max-credit WITH FRAME cust
          TITLE "Customers".
  DOWN WITH FRAME cust.
END.

UP FRAME-DOWN(cust) / 2 WITH FRAME cust.
VIEW FRAME instruct.

REPEAT WITH FRAME cust:
  CHOOSE ROW cust.name KEYS ans AUTO-RETURN NO-ERROR
         WITH FRAME cust.
  COLOR DISPLAY NORMAL cust.name WITH FRAME cust.
  IF ans = "1" THEN SCROLL UP.
  ELSE IF ans = "2" THEN SCROLL FROM-CURRENT UP.
  ELSE IF ans = "3" THEN SCROLL DOWN.
  ELSE IF ans = "4" THEN SCROLL FROM-CURRENT DOWN.
  ELSE IF ans = "5" THEN SCROLL FROM-CURRENT.
  VIEW FRAME cust.
  ans = "".
END.
```

This procedure displays customer information and lets you try each scrolling option from a menu of selections.

r-chose1.p

```
DEFINE VARIABLE counter AS INTEGER.
DEFINE VARIABLE oldchoice AS CHARACTER.

FORM customer.cust-num customer.address customer.city
     customer.zip WITH FRAME cust-frame SCROLL 1
     5 DOWN ATTR-SPACE.

FIND FIRST customer.
REPEAT counter = 1 to 5:
   DISPLAY cust-num address city zip WITH FRAME
           cust-frame.
   DOWN WITH FRAME cust-frame.
   FIND NEXT customer NO-ERROR.
   IF NOT AVAILABLE customer
   THEN LEAVE.
END.
UP WITH FRAME cust-frame.
oldchoice = "".

REPEAT:
   STATUS DEFAULT "Enter C to create, D to delete".
   CHOOSE ROW customer.cust-num NO-ERROR
               WITH FRAME cust-frame.
   COLOR DISPLAY NORMAL customer.cust-num
              WITH FRAME cust-frame.
   /* After choice */
   If frame-value = ""
   THEN NEXT.
   /*Force user to press END or move cursor to valid line*/
   IF frame-value <> oldchoice
   THEN DO:
      oldchoice = frame-value.
      FIND customer WHERE cust-num = INTEGER(frame-value).
   END.

   /* React to moving cursor off the screen. */

   IF LASTKEY = KEYCODE("CURSOR-DOWN")
   THEN DO:
      FIND NEXT customer NO-ERROR.
      IF NOT AVAILABLE customer
      THEN FIND FIRST customer.
      DOWN WITH FRAME cust-frame.
      DISPLAY cust-num address city zip
              WITH FRAME cust-frame.
      NEXT.
  END.
```

(Continued)

r-chosel.p (Continued)

```
IF LASTKEY = KEYCODE("CURSOR-UP")
THEN DO:
  FIND PREV customer NO-ERROR.
  IF NOT AVAILABLE customer
  THEN FIND LAST customer.
  UP WITH FRAME cust-frame.
  DISPLAY cust-num address city zip
          WITH FRAME cust-frame.
  NEXT.
END.

/*CHOOSE selected a valid key.  Check which key. */

IF LASTKEY = KEYCODE("C")
THEN DO:               /* Open a space in the frame. */
  SCROLL FROM-CURRENT DOWN WITH FRAME cust-frame.
  CREATE customer.
  UPDATE cust-num address city zip
         WITH FRAME cust-frame.
  oldchoice = INPUT cust-num.
  NEXT.
END.

IF LASTKEY = KEYCODE("D")
THEN DO:    /* Delete as customer from the database. */
  DELETE customer.
  FIND NEXT customer NO-ERROR.
  /* Move to correct position in database. */
  IF NOT AVAILABLE customer
  THEN DO:
     FIND FIRST customer NO-ERROR.
     IF NOT AVAILABLE customer
     THEN DO:
       CLEAR FRAME cust-frame.
       UP WITH FRAME cust-frame.
       NEXT.
     END.
   END.
```

(Continued)

r-chosel.p (Continued)

```
      IF frame-line(cust-frame) = frame-down(cust-frame)
      /* If last screen line deleted. */
      THEN DO:
        DISPLAY cust-num address city zip
                WITH FRAME cust-frame.
        NEXT.
      END.

      SCROLL FROM-CURRENT WITH FRAME cust-frame.
      REPEAT counter = 1 TO 100
      WHILE frame-line(cust-frame) < frame-down(cust-frame):
        FIND NEXT customer NO-ERROR.
        IF NOT AVAILABLE customer
        THEN DO.
           FIND FIRST customer NO-ERROR.
           IF NOT AVAILABLE customer
           THEN LEAVE.
        END.
        DOWN WITH FRAME cust-frame.
        IF INPUT cust-num = ""
        THEN DISPLAY cust-num address city zip
                     WITH FRAME cust-frame.
      END.
      UP counter - 1 WITH FRAME cust-frame.
      oldchoice = INPUT cust-num.
    END.
  END.
  STATUS DEFAULT.
```

This procedure creates a scrolling frame of five fields. The frame displays the cust-num, address, city, and zip for each customer. The status default message displays "Enter C to create, D to delete" as long as the procedure is running. You use arrow keys to move the highlighted cursor bar through the database, and to add or delete customers from the database. The CHOOSE statement lets you easily create this style menu. See the CHOOSE reference page for more information.

The SCROLL statement controls the scrolling action in the frame when you create and delete customers. You add a customer to the database by entering "C". Create opens a line in the frame and the SCROLL statement moves data below the line down. Then you type the new customer information into the frame. You enter "D" to delete a customer from the database. When you delete a customer, SCROLL moves the rows below the deleted customer row up into the empty line.

You can perform the same function as the previous example with fewer statements if you do not use the SCROLL statement. You can substitute the procedure segment below in the r-chose1.p procedure to perform the delete function.

r-chose2.p

```
                     .
                     .
                     .
IF LASTKEY = KEYCODE("D")
THEN DO:    /* Delete a line from the frame. */
  DELETE customer.
  REPEAT counter = 1 TO 100 WHILE frame-line(cust-frame)
         <= frame-down(cust-frame).
    FIND NEXT customer NO-ERROR.
    IF AVAILABLE customer
    THEN DISPLAY cust-num address city zip
                 WITH FRAME cust-frame.
    ELSE CLEAR FRAME cust-frame.
    DOWN WITH FRAME cust-frame.
  END.
  UP counter - 1 WITH FRAME cust-frame.
  oldchoice = INPUT cust-num.
END.
                     .
                     .
                     .
```

You can see the entire r-chose2.p procedure on-line. The example above shows only the portion that is different from the r-chose1.p procedure.

The following procedure, r-cuhelp.p, was written to provide help for the cust-num field when a user presses . It displays five customer names and numbers and permits a user to scroll up ( ), scroll down ( ), or leave ( ).

r-cuhelp.p

```
FORM customer.cust-num customer.name
WITH FRAME cust-frame 5 DOWN ROW 10 CENTERED
   OVERLAY TITLE " Available Customers ".

REPEAT WHILE FRAME-LINE(cust-frame) <= FRAME-DOWN(cust-frame):
   FIND NEXT customer.
   DISPLAY customer.cust-num customer.name WITH FRAME cust-frame.
   DOWN WITH FRAME cust-frame.
END.

UP 5 WITH FRAME cust-frame.

REPEAT:
   CHOOSE ROW customer.cust-num NO-ERROR WITH FRAME cust-frame.
   FIND customer WHERE customer.cust-num = INPUT customer.cust-num.
   IF KEYFUNCTION(LASTKEY) = "CURSOR-UP" THEN DO:
      FIND PREV customer NO-ERROR.
      IF AVAILABLE customer THEN DO:
           SCROLL DOWN WITH FRAME cust-frame.
           DISPLAY customer.cust-num customer.name WITH FRAME
              cust-frame.
      END.
   END.
   ELSE
   IF KEYFUNCTION(LASTKEY) = "CURSOR-DOWN" THEN DO:
      FIND NEXT customer NO-ERROR.
      IF AVAILABLE customer THEN DO:
           SCROLL UP WITH FRAME cust-frame.
           DISPLAY customer.cust-num customer.name WITH FRAME
              cust-frame.
      END.
   END.
   ELSE
   IF KEYFUNCTION(LASTKEY) = "RETURN" THEN DO:
       FRAME-VALUE = FRAME-VALUE.
       HIDE FRAME cust-frame.
       RETURN.
   END.
END.
```

SEE ALSO CHOOSE Statement, Frame Phrase

SDBNAME Function

SDBNAME accepts as a parameter either an integer expression or a character expression. If the parameter resolves to a currently connected non-PROGRESS database (also known as a non-PROGRESS sub-schema), then the SDBNAME function returns the logical name of the schema holder database containing the non-PROGRESS sub-schema. If the parameter resolves to a currently connected PROGRESS database, the SDBNAME function returns the logical name of this database (in this instance performing identically to the LDBNAME function).

SYNTAX

$$\text{SDBNAME}\left\{\begin{array}{l}\textit{(integer-expression)}\\ \textit{(logical-name)}\\ \textit{(alias)}\end{array}\right\}$$

integer-expression

If the parameter supplied to SDBNAME is an integer expression, and there are, for example, three currently connected databases, then SDBNAME(1), SDBNAME(2), and SDBNAME(3) return the logical names of their respective schema holder databases according to the rule supplied above. Also, continuing the same example of three connected databases, SDBNAME(4), SDBNAME(5), etc., return the ? value.

logical-name or *alias*

These forms of the SDBNAME function require a quoted character string or a character expression as a parameter. If the parameter is the logical name of a connected database or an alias of a connected database, then the logical name of the schema holder database is returned according to the rule supplied above. Otherwise, the ? value is returned.

EXAMPLE

r-sdbnm.p

```
DEFINE VARIABLE i AS INTEGER.
REPEAT i=1 TO NUM-DBS:
   DISPLAY SDBNAME(i)
   SDBNAME(i) = LDBNAME(i)
   FORMAT "SCHEMA-HOLDER/SUB-SCHEMA"
   COLUMN-LABEL "Gateway!Classification".
END.
```

This procedure displays schema-holder databases, if applicable, for all connected databases.

SEE ALSO CONNECT, DISCONNECT, CREATE ALIAS, and DELETE ALIAS statements; CONNECTED, LDBNAME, PDBNAME, DBTYPE, DBRESTRICTIONS, GATEWAYS, FRAME-DB, and NUM-DBS functions.

SEARCH Function

Searches the directories and libraries defined in the PROPATH environment variable for a file. If the file is in your working directory, returns the name of that file. If the file is not in your working directory, SEARCH concatenates the filename to the directory path listed in your PROPATH and returns the resultant pathname. If SEARCH does not find the file, it returns an unknown value (?).

SYNTAX

```
SEARCH( opsys-file )
```

opsys-file

A character expression (a constant, field name, variable or any combination of these that results in a character value) whose value is the name of the file for which you want to search. The name may include a complete or partial directory path. If *opsys-file* is a constant string, you must enclose it in quotation marks (" ").

EXAMPLE

```
r-search.p

DEFINE VARIABLE fullname AS CHARACTER
  FORMAT "x(55)".
DEFINE VARIABLE filename AS CHARACTER
  FORMAT "x(20)".

REPEAT:
  UPDATE filename HELP
    "Try entering 'help.r' or 'dict.r'"
    WITH FRAME a SIDE-LABELS CENTERED.
  fullname = SEARCH(filename).
  IF fullname = ?
  THEN DISPLAY "UNABLE TO FIND FILE" filename
    WITH FRAME b ROW 6 CENTERED NO-LABELS.
  ELSE DISPLAY "Fully Qualified Path Name Of:"
    filename SKIP(2) "is:" fullname
    WITH FRAME c ROW 6 NO-LABELS CENTERED.
END.
```

In this procedure, the SEARCH function returns the fully qualfied path name of the filename entered if it is not in the current working directory. If SEARCH cannot find the file, it returns an unknown value (?). The procedure displays either the fully qualified path name or a message indicating that the file could not be found.

NOTES

- Use the SEARCH function to ensure that procedures that get input from external data files are independent of specific directory paths. The files must be in one of the directories or libraries defined in the PROPATH environment variable.

- If you provide a fully qualified path name, SEARCH checks whether the file exists. SEARCH does not have to search the directories in PROPATH.

- When you search for a file that is in a library, SEARCH returns the file's pathname in the form *path-name<<member-name>>*, where *path-name* is the pathname of the library and *member-name* is the name of the file. The brackets << >> indicate that the file is a member of a library. For example, in the path `/usr/apps.pl<<proc1.r>>`, `proc1.r` is the name of the file in the library `apps.pl`.

  The LIBRARY and MEMBER functions use the special syntax to return, respectively, the library name and *member-name* of the file in the library.

SEEK Function

Returns the offset (in bytes) of the file pointer in an ASCII file. You define a procedure variable to hold the offset value and later position the file to that offset.

SYNTAX

$$\text{SEEK (}\left\{\begin{array}{l}\text{INPUT}\\\text{OUTPUT}\\\textit{name}\end{array}\right\}\text{)}$$

INPUT

If you specify INPUT, the SEEK function returns the current position of the file pointer in the unnamed input stream.

OUTPUT

If you specify OUTPUT, the SEEK function returns the current position of the file pointer in the unnamed output stream.

name

If you specify SEEK(*name*), the SEEK function returns the current position of the file pointer in the named input or output stream. The stream must be associated with an open file, or SEEK returns the unknown value (?).

EXAMPLES

```
r-seek1.p

DEFINE VARIABLE itemno LIKE item.item-num.
DEFINE VARIABLE itdesc LIKE idesc.
DEFINE VARIABLE m-pos AS INT.

SET itemno LABEL
  "Select a record number to position the output file"
    WITH SIDE-LABELS.

OUTPUT TO test.fil.
FIND item WHERE itemno = item-num.
➡ IF item-num = itemno THEN m-pos = SEEK(OUTPUT) .
  EXPORT item-num idesc.
OUTPUT CLOSE.

INPUT FROM test.fil.
SEEK INPUT TO m-pos.
SET itemno itdesc WITH FRAME d2.
INPUT CLOSE.
```

This procedure illustrates how the SEEK function can be used to access data in an ASCII file. Using SEEK in this way allows you to index into a non-indexed file.

In the example, you are prompted to select an item number to position the output file. When a record is found with that item number, the SEEK function returns the offset into the variable `m-pos`. The value for `m-pos` is the current value of the file pointer. The SEEK statement uses the value in `m-pos` to position the file pointer in the unnamed input stream.

NOTES

- The first byte in a file is byte 0.
- You cannot use the SEEK function with the INPUT THROUGH statement, the INPUT-OUTPUT THROUGH statement, or the OUTPUT THROUGH statement. When used with one of these statements, the SEEK function returns the unknown value (?) .
- After you assign the value of the SEEK function to a procedure variable, you can use that value to reposition the file in the event of an error.

SEE ALSO INPUT FROM Statement, OUTPUT TO Statement, SEEK Statement, DEFINE STREAM Statement, Chapter 9 of the *Programming Handbook*

SEEK Statement

Positions the file pointer to a user-defined offset (in bytes) in an ASCII file. This statement does not require that the file be closed and reopened.

SYNTAX

```
       ⎧ INPUT         ⎫       ⎧ expression ⎫
SEEK   ⎨ OUTPUT        ⎬  TO   ⎨            ⎬
       ⎩ STREAM stream ⎭       ⎩ END        ⎭
```

INPUT

If you specify INPUT, the SEEK statement positions the file pointer in the unnamed input stream.

OUTPUT

If you specify OUTPUT, the SEEK statement positions the file pointer in the unnamed output stream.

STREAM *stream*

If you specify STREAM *stream*, the SEEK statement positions the file pointer in the named input or output stream. If you do not name a stream, PROGRESS uses the unnamed stream.

TO *expression*

An expression (a constant, field name, variable name, or any combination of these) whose value is an integer that indicates the byte location to position the file pointer. If *expression* equals "0", the file pointer is positioned to the first byte in the file. If you want to position the pointer to the last byte in the file, but you do not know the offset, use END.

END

Positions the pointer to the last byte in the file.

EXAMPLES

```
                                                          r-seek.p

 /* r-seek.p */
 /* This procedure seeks to the end-of-file, collects
    the seek address, and writes a record. The record
    is subsequently retrieved using the
    SEEK statement on the stashed seek address.
 */
   DEFINE VAR savepos AS INT.
   DEFINE VAR c AS CHAR FORMAT "x(20)".
   OUTPUT TO seek.out APPEND NO-ECHO.
   savepos = SEEK(OUTPUT).
   PUT UNFORMATTED "abcdefg" SKIP.
   OUTPUT CLOSE.
   INPUT FROM seek.out NO-ECHO.
   SEEK INPUT TO savepos.
   SET c.
   DISPLAY c.
   INPUT CLOSE.
```

Since ASCII file formats differ on each machine, the SEEK function does not necessarily return a number that is meaningful to anyone but the SEEK statement itself. With the exception of SEEK to 0 or SEEK TO END, any address used in the SEEK statement is only guaranteed to behave consistently if the address was previously derived from the SEEK function. Therefore, expressions such as SEEK to SEEK (INPUT) - n may not are work the same on each operating system. Record delimiters must be new-lines on UNIX, and carriage-return / line-feed pairs on all others.

NOTES

- The SEEK statement does not work with named streams identified in the INPUT-THROUGH, OUTPUT-THROUGH, or INPUT-OUTPUT-THROUGH statements.
- Expressions such as SEEK to SEEK (INPUT) - n are not guaranteed to work the same on each operating system.

SEE ALSO INPUT FROM, OUTPUT TO Statement, SEEK Function, DEFINE STREAM Statement, Chapter 9 of the *Programming Handbook*

SELECT Statement (SQL)

Retrieves and displays data from a table. Refer to Chapter 15 of the *Programming Handbook* for detailed information about the SELECT statement.

SYNTAX

```
SELECT[ ALL | DISTINCT ] { * | column-list } [ INTO variable-list]
    FROM { table-name [ range-variable ] } [,...]
    [ WHERE search-condition ]
    [ GROUP BY column-list ]
    [ HAVING search-condition ]
    [ ORDER BY { { column-name | n } [ ASC | DESC ] } [,...] ]
```

ALL
: Retrieves all values in the specified columns. ALL is the default.

DISTINCT
: Retrieves only unique values in the specified columns.

*
: Indicates all columns of the specified table(s).

column-list
: The names of the columns you want to retrieve.

INTO *variable-list*
: Lists the local program variables to receive the column values (singleton SELECT).

FROM *table-name*
: The name of the table from which you want to select data.

[*range-variable*]
: An alias for the table name.

WHERE *search-condition*
: Specifies search conditions or join conditions for the data you want to select. The search condition is one or more logical expressions connected by a logical operator (AND, OR, or NOT).

GROUP BY *column-list*
Groups rows that have the same value for the specified column or columns. The GROUP BY clause produces a single row for each group of rows.

HAVING *search-condition*
Specifies one or more qualifying conditions, normally for the GROUP BY clause. The HAVING clause allows you to exclude groups from the query results.

ORDER BY
Sorts the query results by the values in one or more columns.

column-name
The name of a column by which you want to sort the query results.

n
An integer that specifies the position of a column or expression in the select list. Use this option to sort on arithmetic expressions included in the select list.

ASC
Sorts the query results in ascending order. Ascending order is the default. Therefore, if you omit both ASC and DESC, the results are sorted in ascending order.

DESC
Sorts the query results in descending order.

The SELECT statement can also have one or more subqueries.

EXAMPLES

```
SELECT DISTINCT loc FROM item.
```

```
SELECT cust-num, name, order-num, odate
    FROM order
    WHERE sdate IS NULL.
```

```
SELECT item-num, cost
    FROM item WHERE cost < All
        (SELECT cost FROM item WHERE loc = 'Bin 29')
            ORDER BY cost DESC.
```

NOTES

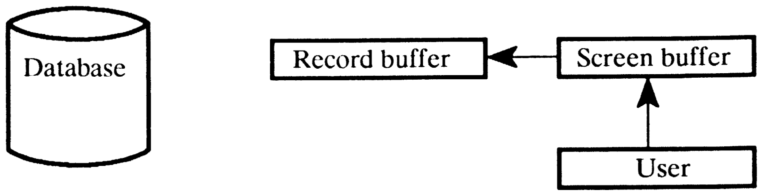
- The SELECT statement can be used in both interactive SQL and embedded SQL.
- In embedded SQL only, you must use a cursor with the SELECT statement, unless you specify the INTO option to perform a singleton SELECT.

SET Statement

Requests input, and then puts the input data in both the screen buffer (frame) and in the specified fields or variables. The SET statement is a combination of the following statements:

PROMPT-FOR	Prompts the user for data and puts that data into the screen buffer.
ASSIGN	Moves data from the screen buffer to the record buffer.

DATA MOVEMENT



SYNTAX

```
SET [ STREAM stream ]
    [ field [ format-phrase ] [ WHEN expression ]     ]
    [ TEXT (field [format-phrase] ... )               ]
    [ field = expression                              ]
    [ constant [ AT n ]                               ] ...
    [          [ TO n ]                               ]
    [ ^                                               ]
    [ SPACE[( n )]                                    ]
    [ SKIP[( n )]                                     ]
    [ GO-ON ( key-label ... ) ]
    [ frame-phrase ]
    [ EDITING-phrase]
```

```
SET [ STREAM stream ] record [EXCEPT field ...] [ frame-phrase ]
```

STREAM *stream*
Specifies the name of a stream. If you do not name a stream, PROGRESS uses the unnamed stream. See the DEFINE STREAM reference page and Chapter 9 of the *Programming Handbook* for more information about streams.

field

The name of the field or variable whose value you want to store in the screen buffer and in the field or variable.

In the case of array fields, array elements with constant subscripts are treated just like any other field. Array fields with no subscripts are expanded as though you had typed in the implicit elements. Array fields with expressions as subscripts are handled as described on the DISPLAY statement reference page.

format-phrase

Specifies one or more frame attributes for a field, variable, or expression. Here is the syntax for *format-phrase*:

```
[ AT n                                                  ]
[ AS datatype                                           ]
[ ATTR-SPACE                                            ]
[ AUTO-RETURN                                           ]
[ BLANK                                                 ]
[ COLON n                                               ]
[ COLUMN-LABEL  label  [ ! label ]...                   ]
[ DEBLANK                                               ] ...
[ FORMAT string                                         ]
[ HELP string                                           ]
[ LABEL string                                          ]
[ LIKE field                                            ]
[ NO-ATTR-SPACE                                         ]
[ NO-LABEL                                              ]
[ TO n                                                  ]
[ VALIDATE ( condition,     msg-expression     )        ]
```

WHEN *expression*

Sets the field only when *expression* has a value of TRUE. *expression* is a field name, variable name, or any combination of these whose value is logical.

field = expression

Indicates that the value of *field* should be determined by evaluating the *expression* rather than having it entered on the screen or from a file. In effect, an assignment statement is embedded within the SET statement.

TEXT

Defines a group of character fields or variables (including array elements) to use automatic word-wrap. The TEXT option works only with character fields. When you insert data in the middle of a TEXT field, PROGRESS wraps data that follows into the next TEXT field, if necessary. If you delete data from the middle of a TEXT field, PROGRESS wraps data that follows up into the empty area. If you enter more characters than the format for the field allows, PROGRESS discards the extra characters. The character fields must have formats of the form "x(*n*)".

A blank in the first column of a line marks the beginning of a paragraph. Lines within a paragraph are treated as a group and will not wrap into other paragraphs.

Table 25 lists the keys you can use within a TEXT field and their actions.

Table 25: Key Actions in a TEXT Field

KEY	ACTION
APPEND-LINE	Combines the line the cursor is on with the next line.
BACK-TAB	Moves the cursor to the previous TEXT field.
BREAK-LINE	Breaks the current line into two lines beginning with the character the cursor is on.
BACK SPACE	Moves the cursor one position to the left and deletes the character at that position. If the cursor is at the beginning of a line, BACK SPACE moves the cursor to the end of the previous line.
CLEAR	Clears the current field and all fields in the TEXT group that follow.
DELETE-LINE	Deletes the line the cursor is on.
NEW-LINE	Inserts a blank line below the line the cursor is on.
RECALL	Clears fields in the TEXT group and returns intial data values for the group.
RETURN	If you are in overstrike mode, moves to the next field in the TEXT group on the screen. If you are in insert mode, the line breaks at the cursor and the cursor is positioned at the beginning of the new line.
TAB	Moves to the field after the TEXT group on the screen. If there is no other field, the cursor moves to the beginning of the TEXT group.

In the procedure that follows, the s-com, or "Order Comments" field is a TEXT field. Run the procedure and enter text in the field to see how the TEXT option works.

```
                                                         r-text.p
DEFINE VARIABLE s-com AS CHARACTER FORMAT "x(40)" EXTENT 5.

FORM "Shipped  :"      order.sdate AT 13 SKIP
     "Misc Info:"      order.misc-info AT 13 SKIP(1)
     "Order Comments:" s-com AT 1
WITH FRAME o-com CENTERED NO-LABELS
     TITLE "Shipping Information".

FOR EACH customer, EACH order OF customer:
  DISPLAY cust.cust-num cust.name order.order-num
          order.odate order.pdate
          WITH FRAME order-hdr CENTERED.
  UPDATE sdate misc-info TEXT(s-com) WITH FRAME o-com.
  s-com = "".
END.
```

constant AT *n*

A constant (literal) value that you want displayed in the frame. *n* is the column at which you want to start the display.

constant TO *n*

A constant value that you want displayed in the frame. *n* is the column at which you want to end the display.

^

You use the caret to tell PROGRESS to ignore an input field when input is being read from a file. Also, the statement

SET ^

will read a line from an input file and ignore that line. This is an efficient way of skipping over lines.

SPACE (*n*)

Identifies the number (*n*) of blank spaces to be inserted after the expression is displayed. *n* can be zero (0). If the number of spaces you specify is more than the spaces left on the current line of the frame, a new line is started and any extra spaces discarded. If you do not use this option or do not use *n*, one space is inserted between items in the frame.

SKIP (*n*)

Identifies the number (*n*) of blank lines to be inserted after the expression is displayed. *n* can be zero (0). If you do not use this option, a line is not skipped between expressions unless they do not fit on one line. If you use the SKIP option but do not specify *n* or if *n* is 0, a new line is started unless it is already at the beginning of a new line.

GO-ON(*key...*)

key... is a list of keyboard key labels. The GO-ON option tells PROGRESS to take the GO action when the user presses any of the keys listed in *key...* . The keys you list are used in addition to keys that perform the GO action by default (e.g. F1 or RETURN on the last field) or because of ON statements. When you list a key in *key...*, you use the keyboard label of that key. For example, if your keyboard has an F2 key and you want PROGRESS to take the GO action when the user presses that key, you use the statement GO-ON(F2). If you list more than one key, separate those keys by spaces, not commas, as in GO-ON(F2 RETURN).

frame-phrase

Specifies the overall layout and processing properties of a frame. Here is the syntax for *frame-phrase*:

```
       [ ACCUM
         ATTR-SPACE
         CENTERED
                 { [ DISPLAY ] color-phrase }
         COLOR   { PROMPT color-phrase      } ...
         COLUMN expression
         n COLUMNS
         DOWN
         expression    DOWN
         FRAME frame
WITH     NO-ATTR-SPACE
         NO-BOX                                   ] ...
         NO-HIDE
         NO-LABELS
         NO-UNDERLINE
         NO-VALIDATE
         OVERLAY
         PAGE-BOTTOM
         PAGE-TOP
         RETAIN n
         ROW expression
         SCROLL n
         SIDE-LABELS
         TITLE [ COLOR color-phrase ] expression
         TOP-ONLY
         WIDTH n                                  ]
```

EDITING-*phrase*

Identifies processing to take place as each keystroke is entered. Here is the syntax of the EDITING-*phrase:*

```
[ label :] EDITING: statement...     END.
```

record

The name of a record buffer. All of the fields in the record, except those with data-type RECID, are processed exactly as if you had set each of them individually. The record you name must contain at least one field.

To use SET with a record in a file defined for multiple databases, you must qualify the record's filename with the database name. See the description of the Record Phrase for more information.

EXCEPT *field*

All fields except those fields listed in the EXCEPT phrase are affected.

EXAMPLES

r-set.p

```
FOR EACH item:
  DISPLAY item-num.
  SET idesc on-hand alloc cost.
END.
```

The r-set.p procedure reads each item record, displays the item-num and lets the user enter information for the idesc, on-hand, alloc, and cost fields. When you run this procedure, notice that it does not display existing values for the idesc, on-hand, alloc, and cost fields.

On each iteration of the block, the FOR EACH statement reads a single record into the item record buffer. The DISPLAY statement moves the item-num from the record buffer to the screen buffer where you can see it. The SET statement prompts for data, stores the data in screen buffers and moves the data to the record buffer, overwriting whatever is already there. Therefore, even though the idesc, on-hand, alloc, and cost fields are put into the item record buffer by the FOR EACH statement, you never see the values for those fields.

r-set2.p

```
FOR EACH customer:
  DISPLAY cust-num name max-credit.
  SET name max-credit
    VALIDATE(max-credit > 0,
             "Invalid max credit.")
    HELP "Enter a positive max-credit.".
  REPEAT:
    CREATE order.
    cust-num = customer.cust-num.
    SET order-num sdate
      VALIDATE(sdate > today,
               "Ship date too early.").
  END.
END.
```

The r-set2.p procedure displays the cust-num, name, and max-credit for a customer and lets you change the max-credit field. The HELP option in the SET statement displays help information at the bottom of the screen when you are changing the max-credit. The VALIDATE option in the SET statement makes sure that the max-credit value is greater than 0. If it isn't, VALIDATE displays the message "Invalid max credit."

After you modify max-credit, the procedure creates an order for the customer and assigns the customer.cust-num value to the cust-num field in the order record. The SET statement lets you enter information for the order-num and sdate fields. The VALIDATE option in the SET statement makes sure that the ship date (sdate) is greater than TODAY.

NOTES

- SET does not move data into the field or variable if there is no data in the corresponding screen field. There is data in a screen field if a DISPLAY of the field was done or if you enter data into the field. If you set a field or variable that has not been DISPLAYed in the frame and key in blanks, then the field or variable is not changed because the screen field is changed only if the data differs from what was in the frame field.

- When PROGRESS compiles a procedure, it designs all the frames used by that procedure. When you run the procedure, the SET statement puts data into those fields.

- In a SET statement, PROGRESS first PROMPTS for all specified fields and then ASSIGNS the values of those fields, moving from left to right. During this left to right pass of the field list, PROGRESS processes imbedded assignments (*field = assignment*) as it encounters them.

- If you are getting input from a device other than the terminal, and the number of characters read by the SET statement for a particular field or variable exeeds the display format for that field or variable, PROGRESS returns an error. However, if you are setting a logical field that has a format of "y/n" and the data file contains a value of "yes" or "no", PROGRESS converts that value to "y" or "n".

- If you type blanks into a field in which data has never been displayed, the ENTERED function returns FALSE and the SET or ASSIGN statement does not update the underlying field or variable. Also, if PROGRESS has marked a field as entered, and the SET statement prompts for the field again, and you do not enter any data, PROGRESS no longer considers the field as entered.

- If you use a single qualified identifer with the SET statement, the compiler first interprets the reference as *dbname.filename*. If the compiler cannot resolve the reference as *dbname.filename*, it tries to resolve it as *filename.fieldname*.

  When using SET to set fields, you must use filenames that are different from field names to avoid ambiguous references. See the description of the Record Phrase for more information.

SEE ALSO DEFINE STREAM Statement, EDITING Phrase, Format Phrase, Frame Phrase, PROMPT-FOR Statement, UPDATE Statement

SETUSERID Function

If the userid and password supplied to the SETUSERID function are in the _User file, SETUSERID returns a TRUE value and assigns the userid to the user. If the userid is not in the _User file or the password is incorrect, SETUSERID returns a FALSE value and does not assign the userid to the user.

SYNTAX

```
SETUSERID (userid, password [ , logical-dbname ])
```

userid

A constant, field name, variable name, or any combination of these that results in a character value that represents the user's userid. If you use a constant, you must enclose it in quotation marks.

password

A constant, field name, variable name, or any combination of these that results in a character value that represents the user's password. If you use a constant, you must enclose it in quotation marks.

logical-dbname

The logical name of the database where you want to check and set your user ID. The logical database name must be a character string enclosed in quotes, or a character expression. If you do not specify this argument, the compiler inserts the name of the database that is connected when the procedure is compiled. If you omit this argument and more than one database is connected, a compiler error results.

EXAMPLE

The login.p procedure is provided with PROGRESS. You see its affects only if security is enabled on your system. That is, you must define userids and passwords for those users who are authorized to access the database.

The login.p procedure uses the SETUSERID function to check the value of the userid and password that a user enters. If the value of the function is false, the procedure allows the user another try. The user has three tries to log in. The first time, the tries variable is 0; tries is 1 the second time, and 2 the third. The third time, tries is greater than 1 and the procedure exits from PROGRESS with the QUIT statement.

login.p

```
/* login.p - prompt user for userid and password and set the userid */

DEFINE VARIABLE id        LIKE _User._Userid.
DEFINE VARIABLE password LIKE _Password.
DEFINE VARIABLE tries    AS INTEGER NO-UNDO.

IF USERID("DICTDB") < > "" OR NOT CAN-FIND(FIRST DICTDB._User)
THEN RETURN.
DO ON ENDKEY UNDO, RETURN: /*return if they hit endkey*/
  /* reset id and password to blank in case of retry */
  id = "".
  password = "".
  UPDATE SPACE(2) id SKIP  password BLANK
    WITH CENTERED ROW 8 SIDE-LABELS ATTR-SPACE
       TITLE " Database " + LDBNAME("DICTDB") + " ".

  IF NOT SETUSERID(id,password,"DICTDB") THEN DO:
    MESSAGE "Sorry, userid/password is incorrect.".
    IF tries > 1 THEN QUIT.  /* only allow 3 tries*/
    tries = tries + 1.
    UNDO, RETRY.
  END.
  HIDE ALL.
  RETURN.
END.
QUIT.
```

NOTES

- Under the following conditions, the SETUSERID function returns a value of FALSE and does not assign a userid to the user:

  - There are no entries in the _User file.

  - There is no _User record with the same userid as the one supplied with the SETUSERID function.

  - The password supplied with the SETUSERID function does not match the password, in the _User file record, of the specified userid.

- When using the SETUSERID function, you receive a compiler error under the following conditions:
  - There is no database connected.
  - The *logical-dbname* argument is omitted, and more than one database is currently connected.
- When specifying the *logical-dbname* argument, you must provide the name of the logical database, not the physical database.
- SETUSERID encodes the *password* argument and then compares the result with the value stored in the _user._password field of the _User file.
- After SETUSERID returns a value of TRUE and assigns a userid to a user:
  - PROGRESS uses that userid when the user compiles procedures.
  - Subsequent uses of the USERID function return the assigned userid.
- If the userid "root" does not exist in the _User file, SETUSERID returns a value of FALSE when supplied with a userid of "root". If the _User file does have a "root" entry, the user who assumes that userid has all the privileges associated with the "root" userid under UNIX.
- You must create a blank userid ("") if you want to set the userid to a null value.
- Tables 26 through 29 show how PROGRESS determines a userid on the supported operating systems.

Table 26: Determining a UNIX Userid

Are there Records in the _User file?	Were the -U and -P Startup Options Supplied?	Userid
NO	YES	Error: -U and -P not allowed unless there are entries in the _User file.
NO	NO	UNIX userid
YES	NO	"" (blank userid)
YES	YES	If the -U useridand -P password match those in the _User file, use that userid. Otherwise, do not assign a userid.

Table 27: Determining a DOS or OS/2 Userid

Are there records in the _User file?	Were the -U and -P start-up options supplied?	Userid
NO	YES	Error: -U and -P not allowed unless there areentries in the _User file.
NO	NO	"" (blank userid)
YES	NO	"" (blank userid)
YES	YES	If the -U userid and -P password match those in the _User file, use that userid. Otherwise, do not assign a userid.

Table 28: Determining a VMS Userid

Are there records in the _User file?	Were the /USER and /PASSWORD options supplied?	Userid
NO	YES	Error: Theseoptions are not allowed unless there are entries in the _User file.
NO	NO	VMS userid
YES	NO	"" (blank userid)
YES	YES	If the /USERID and /PASSWORD match those in the _User file, use that userid. Otherwise, do not assign a userid.

Table 29: Determining a BTOS/CTOS Userid

Are there records in the _User file?	Were the -U and -P start-up options supplied?	Userid
NO	YES	Error: -U and -P not allowed unless there are entries in the _User file.
NO	NO	BTOS/CTOS user name
YES	NO	"" (blank userid)
YES	YES	If the BTOS/CTOS userid and password parameters match those in the _User file, use that userid. Otherwise, do not assign a userid.

SEE ALSO CONNECT Statement, USERID Function, Chapter 11 of the *Programming Handbook*

SQRT Function

Returns the square root (as a decimal value) of an expression you specify.

SYNTAX

SQRT(*expression*)

expression

A numeric expression (a constant, field name, variable name, or any combination of these that results in a numeric value). If the value of the expression is negative, SQRT returns the unknown value (?).

EXAMPLE

r-sqrt.p

```
DEFINE VARIABLE num AS INTEGER FORMAT ">,>>9"
  LABEL "Enter a number between 1 and 9,999".

REPEAT WITH SIDE-LABELS CENTERED TITLE
            "SQUARE ROOT GENERATOR" COLUMN 20
            1 DOWN.
  DISPLAY SKIP(2).
  SET num SKIP(2).
  DISPLAY "The square root of " + string(num) +
          " is" FORMAT "x(27)"
          SQRT(num) FORMAT ">>>.9999".
END.
```

This procedure prompts for a number and then displays the square root of that number.

STATUS Statement

Specifies the text that appears on the bottom (status) line of the terminal screen. PROGRESS displays the following default messages on that line:

- While the user is running a procedure, the status message is all blanks.
- When the user has the opportunity to enter data into a frame field, the status message is "Enter data or press F4 to end."
- When a procedure reaches a PAUSE statement, the status message is "Press space bar to continue."

SYNTAX

$$\text{STATUS}\left\{\begin{array}{l}\text{DEFAULT [} \textit{expression} \text{]}\\ \text{INPUT}\left[\begin{array}{l}\text{OFF}\\ \textit{expression}\end{array}\right]\end{array}\right\}$$

DEFAULT *expression*
Replaces the default status message when a user is running a procedure (the default status message is all blanks). *expression* must be character and must be enclosed in quotes if it is a constant. If you do not specify an *expression*, PROGESS resets the STATUS DEFAULT line to its original state. The STATUS DEFAULT is a maximum of 63 characters.

INPUT OFF
Tells PROGRESS not to display an input status message.

INPUT *expression*
Replaces the default status message when a user is entering data into a frame field. The *expression* must be character and must be enclosed in quotes if it is a constant. If you do not specify an *expression*, PROGRESS resets the STATUS INPUT line to its original state.

EXAMPLE

```
                                                r-status.p
➡ STATUS DEFAULT
    "All Around Sports Order Processing System".
➡ STATUS INPUT "Enter data, or use the "
               + KBLABEL("end-error")
               + " key to exit".

  FOR EACH customer:
    DISPLAY name.
    FOR EACH order OF customer:
      UPDATE order-num pdate odate sdate.
    END.
    UPDATE max-credit.
  END.
```

This procedure replaces the default status messages with two other messages.

NOTES

- After you use either the STATUS DEFAULT, STATUS INPUT OFF, or STATUS INPUT statement during a session, that statement is in effect for all the procedures run in that session unless it is overridden by other STATUS statements in those procedures, or until you return to the editor.
- You cannot use the STATUS statement to change the default status messages displayed while you are in the PROGRESS editor.
- You can use the PAUSE statement to override the default status message displayed when PROGRESS encounters a PAUSE statement.

SEE ALSO MESSAGE Statement, PAUSE Statement

STOP Statement

Stops processing a procedure, backs out the active transaction, and returns to the start-up procedure or to the PROGRESS editor.

SYNTAX

```
STOP
```

EXAMPLE

r-stop.p

```
DEFINE VARIABLE ans AS LOGICAL.
FOR EACH customer:
  DISPLAY cust-num name.
  UPDATE max-credit.
  ans = NO.
  MESSAGE
    "Stopping now undoes changes to this record".
  MESSAGE
    "Do you want to stop now?" UPDATE ans.
  IF ans THEN STOP.
END.
```

In any procedure, the outermost block that updates the database is the system transaction. Here, the first iteration of the FOR EACH block starts a system transaction. The transaction ends when that iteration ends. Another transaction starts at the start of the next iteration. After you update the max-credit field, you are asked if you want to STOP. If you say yes, the STOP statement stops the procedure and undoes any database modifications made in that transaction.

NOTES

- The STOP statement stops all procedures you are running.
- If you use the Startup Procedure (-p) option to start the PROGRESS session, and if the start-up procedure is still active, STOP restarts the procedure.
- A terminal user can do the same thing by pressing STOP. This is CTRL-BREAK (DOS and OS/2), usually CTRL-C (UNIX and VMS), ACTION-CANCEL (BTOS/CTOS) and is dependent on your terminal and system configuration.
- If you press STOP while running applhelp.p, PROGRESS returns to the editor.

SEE ALSO QUIT Statement, and Chapter 2 of the *Programming Handbook* for more information on special keys.

STRING Function

Converts a value of any data type into a character value.

SYNTAX

```
STRING(source [ , format ] )
```

source

An expression (a constant, field name, variable name, or any combination of these that results in a value of any data type) of any data type that you want to convert to a character value.

format

The format you want to use for the new character value. This format must be appropriate to the data type of *source*. If you do not use this argument, PROGRESS uses EXPORT format. This is useful if you want to produce left-justified numbers. See Chapter 4 of the *Programming Handbook* for information on data formats.

EXAMPLE

r-string.p

```
DISPLAY SKIP(2) "The time is now"
  STRING(TIME,"HH:MM AM") SKIP(2)
  WITH NO-BOX NO-LABELS CENTERED.

FOR EACH customer:
  DISPLAY name AT 1 address AT 31
          city + ", " + st + " " +
          STRING(zip, "99999") FORMAT "x(22)"
          AT 31 WITH NO-BOX NO-LABELS CENTERED.
END.
```

The TIME function returns the number of seconds since midnight. The first DISPLAY statement in this procedure uses the STRING function to convert that value into hours and minutes. In that STRING function, TIME is the value and “HH:MM AM” is the format used to display the result of the STRING function.

The second DISPLAY statement displays some customer information. It uses the concatenation (+) operator to join together the values of the city, st, and zip fields. If these fields were not joined together, the spacing would be different for each customer address depending on the length of the city name. Concatenating the city and st fields is simple because they are both character fields. However, concatenating the zip field presents a problem because the zip field is an integer rather than a character data type. The STRING function converts the zip field into a character field using the format "99999." It can then be concatenated with the city and st fields.

Any time you concatenate character fields, you are telling PROGRESS to create a new character field, at least for the duration of the procedure. The default display format for character expressions such as that resulting from the concatenation is "x(8)". This means that PROGRESS would allow only 8 spaces for displaying the concatenation of the city, st, and zip fields. The FORMAT "x(22)" option overrides that default "x(8)" format, telling PROGRESS to set aside 22 spaces for displaying the concatenation of the city, st, and zip fields.

NOTES

- If *source* is an integer and *format* begins "HH:MM" or "HH:MM:SS", STRING formats the *source* as a time. If the hour is greater than or equal to 12 and there is an "A" or an "a" in *format*, STRING subtracts 12 from the hour and converts the "A" or the "a" to a "P" or "p" (for AM and PM). The hour 0 is treated as 12 AM, and noon is treated as 12 PM. If you use AM/PM format, HH is replaced by a leading blank and a digit if the hour is between 0 and 9.

  If seconds (SS) are not in the format, then the time is truncated (not rounded) to hours and minutes.

SEE ALSO DECIMAL Function, INTEGER Function

SUBSTRING Function

Extracts a portion of a character string from a field or variable.

SYNTAX

```
SUBSTRING( source, position [ , length ] )
```

source
A character expression (a constant, field name, variable name, or any combination of these that results in a character value) from which you want to extract a string of characters.

position
An integer expression (a constant, field name, variable name, or any combination of these that results in an integer value) that indicates the position of the first character you want to extract from *source.*

length
An integer expression that indicates the number of characters you want to extract from *source.* If you do not use the *length* argument, SUBSTRING uses the entire *source.*

EXAMPLE

The r–substr.p procedure uses the SUBSTRING function to create invoice numbers. You supply a starting invoice number. The first SUBSTRING function produces the first two characters of today's date; the second SUBSTRING function produces the last two characters of today's date. The procedure concatenates these four characters to a hyphen and the number you entered to produce an invoice number.

r-substr.p

```
DEFINE VARIABLE inv-num AS CHARACTER
  FORMAT "x(11)" LABEL "Invoice Number".
DEFINE VARIABLE snum AS INTEGER
  FORMAT "9999" LABEL "  Starting Order Number".
DEFINE VARIABLE enum LIKE snum
  LABEL "  Ending Order Number".
DEFINE VARIABLE num LIKE snum
  LABEL "Starting Invoice Number".

UPDATE "     Creating Invoices" SKIP(2) snum
     SKIP(1) enum SKIP(2) num SKIP(2)
     WITH SIDE-LABELS CENTERED NO-BOX.

FOR EACH order WHERE order.order-num >= snum
  AND order.order-num <= enum:
    inv-num = SUBSTRING(STRING(TODAY),1,2) +
              SUBSTRING(STRING(TODAY),7,2) +
              " - " + STRING(num,"9999").
  DISPLAY order-num inv-num WITH CENTERED.

 /* Do creation and printing of invoices here */

  num = num + 1.

END.
```

SEE ALSO SUBSTRING Statement, OVERLAY Function

SUBSTRING Statement

Replaces characters in a field or variable with an expression you specify.

SYNTAX

```
SUBSTRING( source, position  [ , length ] ) = expression
```

source

A field or variable in which you want to store *expression*.

position

An integer expression (a constant, field name, variable name, or any combination of these that results in an integer value) that indicates the position in *source* at which you want to start storing *expression*. If *position* is longer than *source*, PROGRESS pads *source* with blanks to equal the length of *position*.

length

An integer expression that indicates the number of positions you want to replace in *source*. If you specify a 0 length, the expression is inserted at the *position* and everything else moves to the right. If you do not use the *length* argument, SUBSTRING puts the entire *expression* into *source*, replacing everything in *source* necessary to make room for *expression*.

expression

A constant, field name, variable name, or any combination of these that results in a character string whose value you want to store in *source*. PROGRESS does not pad or truncate *expression*.

EXAMPLE

```
r-sub.p

DEFINE VARIABLE rtext AS CHARACTER FORMAT "x(50)".
DEFINE VARIABLE orig AS CHARACTER FORMAT "x(31)".
DEFINE VARIABLE strt AS INTEGER FORMAT ">9".
DEFINE VARIABLE leng AS INTEGER FORMAT ">9".

orig = "Now is the time to use PROGRESS".
DISPLAY orig WITH CENTERED TITLE "Original Text" NO-LABEL.
REPEAT:
  rtext = orig.
  UPDATE strt LABEL "START" leng LABEL "LENGTH".
  SUBSTRING(rtext,strt,leng) = "XXXXXXXXX".
  DISPLAY rtext LABEL "WORD" WITH CENTERED.
END.
```

The r-sub.p procedure uses the SUBSTRING statement to replace a segment of text with the expression in the SUBSTRING statement (XXXXXXXXX). The procedure first displays the text you can work with in the Original Text frame. Then the procedure prompts you for the start position (numeric) of the replacement and the length (numeric) of the replacement. Under the heading WORD, you see the revised text.

SEE ALSO OVERLAY Function, SUBSTRING Function

TERMINAL Function

On DOS and OS/2 systems, returns the value BW80, CO80, or MONO, depending on the monitor type. On UNIX and BTOS/CTOS systems, TERMINAL returns the value of the $TERM variable. On VMS systems, it returns the value of the PROTERM logical. If the PROTERM logical has not been set, the TERMINAL function returns the value of the TERM logical. In batch mode, the TERMINAL function returns a null string.

SYNTAX

```
TERMINAL
```

EXAMPLE

```
r-term.p

   MESSAGE "You are currently using a"
→           TERMINAL "terminal".
```

This one line procedure displays a message that tells you what kind of terminal you are using.

SEE ALSO TERMINAL Statement

TERMINAL Statement

Change terminal type during program execution. On UNIX and BTOS/CTOS systems, change the value of the TERM environment variable. On VMS systems, change the value of the PROTERM logical variable (or TERM if PROTERM has not been set).

SYNTAX

```
TERMINAL = termid.
```

termid

A terminal type string. *Termid* can also be an expression. An error message appears if *termid* is not defined in the PROTERMCAP file. One exception: *termid* can be the word `TERMINAL`. The line `TERMINAL=TERMINAL` reinitializes the terminal.

EXAMPLE

r-seterm.p

```
FOR EACH customer:
        DISPLAY customer.
END.

TERMINAL = "wy60w".
OUTPUT TO TERMINAL PAGED.
FOR EACH customer:
        DISPLAY customer WITH WIDTH 132.
END.

OUTPUT CLOSE.
TERMINAL = "wy60".
DISPLAY "Back to 80 columns." WITH CENTERED.
```

In this example, the terminal screen width is switched from 80 columns to 132 columns and back again.

NOTES

- TERMINAL does not change the physical characteristics of a terminal. You must supply a valid terminal type for the existing terminal state.

- If a subprocedure uses a frame, the frame is composed with the width that was in effect when the subprocedure was compiled. Changing the width (terminal type) outside the scope of that procedure will not change the frame width inside the procedure unless it is recompiled. The following sequence of statements does not work as intended, because `subproc.p` is not recompiled before its second execution.

r-setrml.p

```
TERMINAL = "wy60w".
RUN subproc.p.
TERMINAL = "wy60".
RUN subproc.p.
DISPLAY "Frame (132) too big for screen (80)" WITH CENTERED.
```

SEE ALSO TERMINAL Function

TIME Function

Returns the number of seconds since midnight (local time). You can use this function together with the STRING function to produce the time in hours, minutes, and seconds.

SYNTAX

```
TIME
```

EXAMPLES

r-time.p

```
DEFINE VARIABLE hour AS INTEGER.
DEFINE VARIABLE minute AS INTEGER.
DEFINE VARIABLE sec AS INTEGER.
DEFINE VARIABLE timeleft AS INTEGER.

/* seconds till next midnight */
timeleft = (24 * 60 * 60) - TIME.
sec = timeleft MOD 60.
/* minutes till next midnight */
timeleft = (timeleft - sec) / 60.
minute = timeleft MOD 60.
/* hours till next midnight */
hour = (timeleft - minute) / 60.
DISPLAY "Time to midnight:" hour minute sec.
```

In r-time.p, the timeleft variable is set to the result of the TIME function subtracted from the number of seconds in a day. From there, the procedure translates this into seconds, minutes, and hours.

r-time2.p

```
DISPLAY STRING(TIME,"HH:MM:SS").
```

This DISPLAY statement displays the current time.

SEE ALSO STRING Function, TODAY Function

TODAY Function

Returns the current system date.

SYNTAX

```
TODAY
```

EXAMPLE

r-today.p

```
DEFINE VARIABLE rptdate AS DATE.

OUTPUT TO PRINTER.

rptdate = TODAY.
FORM HEADER rptdate "Customer List" AT 34
  "Page" AT 66 PAGE-NUMBER FORMAT ">>>9" SKIP(2)
  WITH NO-BOX PAGE-TOP.
VIEW.

FOR EACH customer:
  DISPLAY name AT 1 address AT 31
          city + ", " + st + " " +
          STRING(zip, "99999") FORMAT "x(22)"
          AT 31 WITH NO-BOX NO-LABELS CENTERED.
END.
```

This procedure prints the date in the first line at the top of each page of a report. Instead of using TODAY directly in the FORM statement, the procedure uses a variable to hold the date. This ensures that the same date appears on all pages of the report even if this procedure runs through midnight.

Because PAGE-TOP frames are reevaluated on every new page, if a variable was not used, a different date would result if the report started before midnight and ended after midnight.

SEE ALSO TIME Function

TRIM Function

Removes leading and trailing spaces in a character string.

SYNTAX

```
TRIM ( expression )
```

expression

A character expression. The *expression* can be defined as a constant, field name, variable name, or any combination of these.

EXAMPLE

```
                                                    r-trim.p
      DEFINE VARIABLE menu AS CHARACTER EXTENT 3.
      DO WHILE TRUE:
        DISPLAY
              "  1. Display Customer Data" @ menu[1] SKIP
              "  2. Display Order Data"    @ menu[2] SKIP
leading       "  3. Exit"                  @ menu[3] SKIP
spaces      WITH FRAME choices NO-LABELS.
        CHOOSE FIELD menu AUTO-RETURN WITH FRAME choices
          TITLE "Demonstration Menu" CENTERED ROW 10.
        HIDE FRAME choices.
        IF TRIM(FRAME-VALUE) BEGINS "1" THEN RUN r-dblnkc.p.
        IF TRIM(FRAME-VALUE) BEGINS "2" THEN RUN r-dblnko.p.
        IF TRIM(FRAME-VALUE) BEGINS "3" THEN LEAVE.
      END.
```

This procedure displays a menu from which you can choose to display customer and order information. The option numbers are displayed with leading spaces. The TRIM function removes the leading spaces so the menu selection can be easily evaluated.

NOTES

- A character string displays with the default format of "x(8)", unless you specify a format or use a statement such as "DISPLAY @ *literal*".
- TRIM can also be used as a keyword in the FORM statement. In that context, it is used to remove leading spaces for fields in the input buffer.

SEE ALSO FORM statement

TRUNCATE Function

Truncates a decimal expression to a specified number of places, returning a decimal value.

SYNTAX

```
TRUNCATE ( expression, precision )
```

expression
: A decimal expression (a constant, field name, variable name, or any combination of these that results in a decimal value) that you want to truncate.

precision
: A non-negative integer expression that indicates the number of decimal places you want in the result of truncating the *expression*.

EXAMPLE

```
r-trunc.p

FOR EACH customer:
  FORM cust-num name max-credit new-max
    LIKE max-credit LABEL "New Max Credit".
  DISPLAY cust-num name max-credit.
  max-credit =
    TRUNCATE((max-credit * 2) / 1000 ,0) * 1000.
  IF max-credit < 500 THEN max-credit = 500.
  DISPLAY max-credit @ new-max.
END.
```

This procedure doubles each customer's max-credit and then truncates that value before rounding it to the nearest thousand dollars.

NOTE

- You can use the TRUNCATE function to treat division as integer division. (e.g. i = TRUNCATE (x / y, 0)).

SEE ALSO ROUND Function

UNDERLINE Statement

Underlines a field or variable, using the next display line for the underline.

SYNTAX

```
UNDERLINE [ STREAMstream ] field... [ frame-phrase ]
```

STREAM *stream*

Specifies the name of a stream. If you do not name a stream, the unnamed stream is used. See the DEFINE STREAM reference page and Chapter 9 of the *Programming Handbook* for more information about streams.

field

The name of the field or variable you want to underline.

frame-phrase

Specifies the overall layout and processing properties of a frame. Here is the syntax of *frame-phrase*:

```
       [ ACCUM                                          ]
       [ ATTR-SPACE                                     ]
       [ CENTERED                                       ]
       [        { [ DISPLAY ] color-phrase }            ]
       [ COLOR  { PROMPT color-phrase      } ...        ]
       [ COLUMN expression                              ]
       [ n COLUMNS                                      ]
       [ DOWN                                           ]
       [ expression DOWN                                ]
       [ FRAME frame                                    ]
WITH   [ NO-ATTR-SPACE                                  ] ...
       [ NO-BOX                                         ]
       [ NO-HIDE                                        ]
       [ NO-LABELS                                      ]
       [ NO-UNDERLINE                                   ]
       [ NO-VALIDATE                                    ]
       [ OVERLAY                                        ]
       [ PAGE-BOTTOM                                    ]
       [ PAGE-TOP                                       ]
       [ RETAIN n                                       ]
       [ ROW expression                                 ]
       [ SCROLL n                                       ]
       [ SIDE-LABELS                                    ]
       [ TITLE [ COLOR color-phrase ] expression        ]
       [ TOP-ONLY                                       ]
       [ WIDTH n                                        ]
```

EXAMPLE

r-underl.p

```
FOR EACH customer BREAK BY st:
  DISPLAY st cust-num name.
  IF LAST-OF(st) THEN UNDERLINE st.
END.
```

This procedure produces a report of customer records, categorized by state. Whenever the last customer for a certain state has been displayed (determined by the LAST-OF function), the UNDERLINE statement underlines the state field.

NOTES

- The UNDERLINE statement is particularly useful for highlighting certain fields or for underlining accumulated values that you calculated using functions other than the automatic aggregate functions supplied with PROGRESS.
- When determining the position within a DOWN frame, the DOWN statement and the UP statement count the line used by an underline.
- Even if the layout of a DOWN frame takes multiple screen lines, the underline takes just one line on the screen.
- For a 1 DOWN frame or single frame, the UNDERLINE does not appear. Instead, PROGRESS clears the frame.

SEE ALSO DEFINE STREAM Statement, Frame Phrase

UNDO Statement

Backs out all modifications to fields and variables made during the current iteration of a block, and indicates what action to take next.

SYNTAX

```
                    [ , LEAVE  [ label2 ] ]
UNDO [ label1 ]     [ , NEXT   [ label2 ] ]
                    [ , RETRY  [ label2 ] ]
                    [ , RETURN            ]
```

label1

The name of the block whose processing you want to undo. If you do not name a block with *label1*, UNDO undoes the processing of the closest transaction or subtransaction block. In determining the closest transaction or subtransaction block, PROGRESS disregards DO ON ENDKEY blocks which do not also have the ON ERROR or TRANSACTION option.

LEAVE *label2*

Indicates that, after undoing the processing of a block, PROGRESS should leave the block you name with *label2*. If you do not name a block with the LEAVE option, PROGRESS leaves the block that was undone. After leaving a block, PROGRESS continues on with any remaining processing in a procedure.

NEXT *label2*

Indicates that, after undoing the processing of a block, PROGRESS should do the next iteration of the block you name with *label2*. If you do not name a block, PROGRESS does the next iteration of the block that was undone.

RETRY *label2*

Indicates that, after undoing the processing of a block, PROGRESS should repeat the same iteration of the block you name with *label2*. If you name a block with *label2* it must be the name of the block that was undone.

RETRY is the default processing if you do not use any of LEAVE, NEXT, RETRY, or RETURN. When a block is retried, any frames scoped to that block are not advanced or cleared. For more information, see Chapter 8 of the *Programming Handbook*.

RETURN

Returns to the calling procedure or, if there was none, to the PROGRESS editor.

EXAMPLE

```
                                                  r-undo.p

DEFINE VARIABLE ans AS LOGICAL.

REPEAT FOR salesrep WITH ROW 7 1 COLUMN 1 DOWN
  CENTERED ON ENDKEY UNDO, LEAVE:
  PROMPT-FOR sales-rep.
  FIND salesrep USING sales-rep NO-ERROR.
  IF NOT AVAILABLE salesrep THEN DO:
    ans = YES.
    MESSAGE "Salesrep record does not exist".
    MESSAGE "Do you want to add a salesrep?"
      UPDATE ans.
    IF ans THEN DO:
      CREATE salesrep.
      ASSIGN sales-rep.
      UPDATE slsname slsrgn slstitle slsquota.
    END.
➤   ELSE UNDO, RETRY.
  END.
  ELSE DISPLAY salesrep.
END.
```

The r-undo.p procedure prompts you for the initials of a salesrep. If the initials match those of an existing salesrep, the procedure displays that salesrep's record. Otherwise, it asks if you want to add another salesrep with the initials you supplied. If you say no, the UNDO statement undoes the work you've done since the start of the REPEAT block and lets you enter another set of initials.

NOTES

- You can also specify UNDO processing for a block by using the ON ERROR and ON ENDKEY phrases with a block statement.
- An UNDO statement that specifies a block that encompasses the current system transaction block has no effect on changes made prior to the start of the system transaction. This includes changes made to variables prior to the beginning of the system transaction.
- If nothing will be different during a RETRY of a block, then the RETRY is treated as a NEXT or a LEAVE. This default action provides protection against infinite loops.

SEE ALSO ON ENDKEY Phrase, ON ERROR Phrase, RETRY Function, Chapter 8 of the *Programming Handbook*

UNIX Statement

Runs a program, UNIX command or UNIX script, or starts a UNIX interactive shell to allow interactive processing of UNIX commands.

SYNTAX

```
UNIX [ SILENT ] [ unix-command         [ argument              ] ... ]
                [ VALUE( expression )  [ VALUE( expression )   ]     ]
```

SILENT

After processing a UNIX statement, PROGRESS pauses and asks you to press the space bar to continue. You can use the SILENT option to eliminate this pause. Use this option only if you are sure that the UNIX program, command, or batch file does not generate any output to the screen.

unix-command

The name of the program, command, or batch file you want to run. If you do not use this option, the UNIX statement invokes the UNIX shell and remains there until you type CTRL-D or the EOF character set by the UNIX stty command.

VALUE*(expression)*

expression is a constant, field name, variable name, or any combination of these, whose value is the name of a program, command, or batch file you want the UNIX statement to use.

argument

One or more literal arguments you want to pass to the program, command, or batch file being run.

VALUE*(expression)*

expression is a constant, field name, variable name, or any combination of these, whose value is an argument you want to pass to the program, command or batch file being run.

EXAMPLE

```
                                                  r-unx.p
   DEFINE VARIABLE proc AS CHARACTER
     FORMAT "x(10)".

   REPEAT:
     DISPLAY "Enter L to list your files"
       WITH ROW 5 CENTERED FRAME a.
     SET proc LABEL "Enter a valid Procedure Name"
       WITH ROW 9 CENTERED FRAME b.
     IF proc = "L" THEN
→      IF OPSYS = "UNIX" THEN UNIX ls.
       ELSE IF OPSYS = "MSDOS" THEN DOS dir.
       ELSE IF OPSYS = "OS2" THEN OS2 dir.
       ELSE IF OPSYS = "BTOS" THEN BTOS
                "[sys]<sys>files.run" files.
       ELSE VMS directory.
     ELSE DO:
       HIDE FRAME a.
       HIDE FRAME b.
       RUN VALUE(proc).
     END.
   END.
```

In r-unx.p, if you enter an "L", PROGRESS runs the DOS dir command, OS2 dir command, VMS directory command, the BTOS [sys] < sys > files.run files command or the UNIX command. If you enter a procedure name, that name is stored in the proc variable. The RUN statement then runs the procedure.

NOTES

- If you are using DOS, OS/2, BTOS/CTOS or VMS and you use the UNIX statement in a procedure, that procedure will compile. The procedure will run as long as flow of control does not pass through the UNIX statement.

- This command does not exit to UNIX and return. It creates a "shell" within PROGRESS to execute the command. Because of this, you cannot use the UNIX statement as a substitute for QUIT.

- You can also access an interactive UNIX shell by selecting option "e" from the main menu of PROGRESS Help.

- When you use the UNIX cp command as a PROGRESS statement, PROGRESS assumes that a period (.) indicates the end of the statement. This causes the cp command to display a message stating that it requires two arguments. For example, PROGRESS uses the period as the end of the statement indicator:

```
UNIX cp usr/myfile
```

To use the period as part of a UNIX command, enclose the command in quotation marks (" "), as in:

```
UNIX "cp usr/myfile ."
```

SEE ALSO DOS Statement, OPSYS Function, VMS Statement, BTOS Statement, OS2 Statement, CTOS Statement.

UP Statement

Explicitly positions the cursor on a new line in a down, or multi-line frame.

SYNTAX

```
UP [ STREAM stream ] [ expression ] [ frame-phrase ]
```

STREAM *stream*

Specifies the name of a stream. If you do not name a stream, the unnamed stream is used. See the DEFINE STREAM reference page and Chapter 9 of the *Programming Handbook* for more information about streams.

expression

The number of occurrences of data in the frame that you want to move up. UP is the same as UP 1, except nothing happens until the next data handling statement that affects the screen. Several UP statements in a row with no intervening displays are treated like a single UP 1. UP 0 does nothing. If *expression* is negative, the result is the same as DOWN *expression*.

```
        [ ACCUM                                                  ]
        [ ATTR-SPACE                                             ]
        [ CENTERED                                               ]
        [       { [ DISPLAY ] color-phrase  }                    ]
        [ COLOR { PROMPT color-phrase       } ...                ]
        [ COLUMN expression                                      ]
        [ n COLUMNS                                              ]
        [ DOWN                                                   ]
        [ expression    DOWN                                     ]
        [ FRAME frame                                            ]
WITH    [ NO-ATTR-SPACE                                          ] ...
        [ NO-BOX                                                 ]
        [ NO-HIDE                                                ]
        [ NO-LABELS                                              ]
        [ NO-UNDERLINE                                           ]
        [ NO-VALIDATE                                            ]
        [ OVERLAY                                                ]
        [ PAGE-BOTTOM                                            ]
        [ PAGE-TOP                                               ]
        [ RETAIN n                                               ]
        [ ROW expression                                         ]
        [ SCROLL n                                               ]
        [ SIDE-LABELS                                            ]
        [ TITLE [ COLOR color-phrase ] expression                ]
        [ TOP-ONLY                                               ]
        [ WIDTH n                                                ]
```

frame-phrase
Specifies the overall layout and processing properties of a frame. Here is the syntax of *frame-phrase*:

EXAMPLE

r-up.p

```
FOR EACH customer:
  UP 2.
  DISPLAY cust-num name address city st.
END.
```

This procedure starts at the bottom of the screen and displays all the customer database records. The default frame for the FOR EACH block is a down frame. The DISPLAY statement uses that frame. Therefore, PROGRESS automatically advances down the screen 1 line after each iteration. You need to say UP 2 rather than UP 1 because there is an automatic DOWN 1 performed on the display frame at the end of each iteration of the FOR EACH block.

NOTES

- When a frame is a down frame, PROGRESS automatically advances to the next frame line on each iteration of the block to which it is scoped. This is true whether you use the DOWN statement or not. If you do not want PROGRESS to do this automatic advancing, name the frame outside of the current block. For more information on frames, see Chapter 7 of the *Programming Handbook.*

- When PROGRESS reaches the top (first) frame line and then encounters an UP statement, it clears the frame and starts at the bottom line of the frame. However, if you are using SCROLL, PROGRESS moves everything in the frame down one row.

SEE ALSO DEFINE STREAM Statement, DOWN Statement, Frame Phrase, SCROLL Statement, Chapter 7 of the *Programming Handbook*

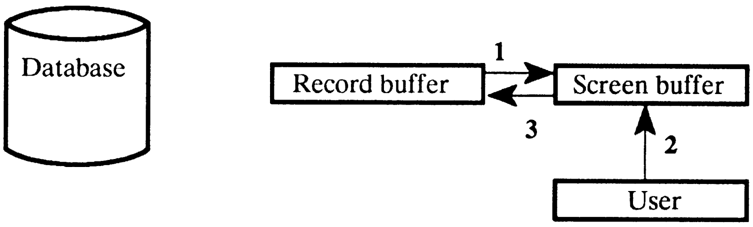
UPDATE Statement

Displays fields or variables, requests input, and then puts the input data in both the screen buffer (frame) and in the specified fields or variables.

The UPDATE statement is a combination of the following statements:

DISPLAY	Moves the values of fields or variables into the screen buffer and displays them.
PROMPT-FOR	Prompts the user for data and puts that data into the screen buffer.
ASSIGN	Moves data from the screen buffer to the record buffer.

DATA MOVEMENT



SYNTAX

```
UPDATE
  [ field [ format-phrase ] [ WHEN expression ]      ]
  [ TEXT ( field [ format-phrase ] ... )              ]
  [ field = expression                                ]
  [ constant [ AT n ]                                 ] ...
  [          [ TO n ]                                 ]
  [ ^                                                 ]
  [ SPACE[( n )]                                      ]
  [ SKIP[( n )]                                       ]
  [ GO-ON ( key-label ... ) ]
  [ frame-phrase ]
  [ EDITING-phrase ]
```

```
UPDATE record [ EXCEPT field ...] [ frame-phrase ]
```

field

The name of the field or variable whose value you want to display, change, and store in the screen and record buffers.

In the case of array fields, array elements with constant subscripts are treated just like any other field. Array fields with no subscripts are expanded as though you had typed in the implicit elements. Array fields with expressions as subscripts are handled as described on the DISPLAY statement reference page.

You can supply values for array elements in the UPDATE statement. For example:

```
UPDATE x[1] = "x".
```

This statement assigns the letter "x" to the first element of array x. If you do not include an array subscript, PROGRESS assigns the value to all elements of the array. For example:

```
UPDATE x = "x".
```

This statement assigns the letter "x" to all elements of the array x.

format-phrase

Specifies one or more frame attributes for a field, variable, or expression. Here is the syntax for *format-phrase*:

```
[ AT n                                        ] ...
[ AS datatype                                 ]
[ ATTR-SPACE                                  ]
[ AUTO-RETURN                                 ]
[ BLANK                                       ]
[ COLON n                                     ]
[ COLUMN-LABEL   label   [ ! label ]...       ]
[ DEBLANK                                     ]
[ FORMAT string                               ]
[ HELP string                                 ]
[ LABEL string                                ]
[ LIKE field                                  ]
[ NO-ATTR-SPACE                               ]
[ NO-LABEL                                    ]
[ TO n                                        ]
[ VALIDATE ( condition, msg-expression )      ]
```

For more information on *format-phrase*, see the Format Phrase reference page.

WHEN *expression*

Updates the field only when *expression* has a value of TRUE. Here, *expression* is a field name, variable name, or any combination of these whose value is logical.

TEXT

Defines a group of character fields or variables (including array elements) to use automatic word-wrap. The TEXT option works only with character fields. When you insert data in the middle of a TEXT field, PROGRESS wraps data that follows into the next TEXT field, if necessary. If you delete data from the middle of a TEXT field, PROGRESS wraps data that follows up into the empty area. If you enter more characters than the format for the field allows, PROGRESS discards the extra characters. The character fields must have formats in the form "x(n)". A blank in the first column of a line marks the beginning of a paragraph. Lines within a paragraph are treated as a group and will not wrap into other paragraphs.

Table 30 lists the keys you can use within a TEXT field and their actions.

Table 30: Key Actions in a TEXT Field

KEY	ACTION
APPEND-LINE	Combines the line the cursor is on with the next line.
BACK-TAB	Moves the cursor to the previous TEXT field.
BREAK-LINE	Breaks the current line into two lines beginning with the character the cursor is on.
BACK SPACE	Moves the cursor one position to the left and deletes the character at that position. If the cursor is at the beginning of a line, BACK SPACE moves the cursor to the end of the previous line.
CLEAR	Clears the current field and all fields in the TEXT group that follow.
DELETE-LINE	Deletes the line the cursor is on.
NEW-LINE	Inserts a blank line below the line the cursor is on.
RECALL	Clears fields in the TEXT group and returns initial data values for the group.
RETURN	If you are in overstrike mode, moves to the next field in the TEXT group on the screen. If you are in insert mode, the line breaks at the cursor and the cursor is positioned at the beginning of the new line.
TAB	Moves to the field after the TEXT group on the screen. If there is no other field, the cursor moves to the beginning of the TEXT group.

In the procedure that follows, the s-com, or "Order Comments" field is a TEXT field. Run the procedure and enter text in the field to see how the TEXT option works.

```
r-text.p

DEFINE VARIABLE s-com AS CHARACTER FORMAT "x(40)" EXTENT 5.

FORM "Shipped  :"      order.sdate AT 13 SKIP
     "Misc Info:"      order.misc-info AT 13 SKIP(1)
     "Order Comments:" s-com AT 1
WITH FRAME o-com CENTERED NO-LABELS
     TITLE "Shipping Information".

FOR EACH customer, EACH order OF customer:
  DISPLAY cust.cust-num cust.name order.order-num
          order.odate order.pdate
          WITH FRAME order-hdr CENTERED.
  UPDATE sdate misc-info TEXT(s-com) WITH FRAME o-com.
  s-com = "".
END.
```

field = *expression*

Indicates that the value of *field* should be determined by evaluating the expression rather than having it entered on the screen or from a file. In effect, an assignment statement is embedded in the UPDATE statement.

constant AT *n*

A constant (literal) value that you want to display in the frame. *n* is the column at which you want to start the display.

constant TO *n*

A constant value that you want to display in the frame. *n* is the column at which you want to end the display.

^

You use the caret to tell PROGRESS to ignore an input field when input is being read from a file. Also, the statement

```
UPDATE ^
```

will read a line from an input file and ignore that line.

SPACE *(n)*

Identifies the number (*n*) of blank spaces to be inserted after the expression is displayed. *n* can be zero (0). If the number of spaces you specify is more than the spaces left on the current line of the frame, a new line is started and any extra spaces discarded. If you do not use this option or do not use *n*, one space is inserted between items in the frame.

SKIP *(n)*

Identifies the number (*n*) of blank lines to be inserted after the expression is displayed. *n* can be zero (0). If you do not use this option, a line is not skipped between expressions unless they do not fit on one line. If you use the SKIP option but do not specify *n* or if *n* is 0, a new line is started unless it is already at the beginning of a new line.

GO-ON(*key...*)

key... is a list of keyboard key labels. The GO-ON option tells PROGRESS to take the GO action when the user presses any of the keys listed in *key...* . The keys you list are used in addition to keys that perform the GO action by default (e.g. F1 or RETURN on the last field) or because of ON statements. When you list a key in *key...*, you use the keyboard label of that key. For example, if your keyboard has an F2 key and you want PROGRESS to take the GO action when the user presses that key, you use the statement GO-ON(F2). If you list more than one key, separate those keys by spaces, not commas.

frame-phrase

Specifies the overall layout and processing properties of a frame. Here is the syntax for *frame-phrase*:

```
       ┌ ACCUM                                                        ┐
       │ ATTR-SPACE                                                   │
       │ CENTERED                                                     │
       │        { [ DISPLAY ] color-phrase  }                         │
       │ COLOR  { PROMPT color-phrase       } ...                     │
       │ COLUMN  expression                                           │
       │ n COLUMNS                                                    │
       │ DOWN                                                         │
       │ expression      DOWN                                         │
       │ FRAME frame                                                  │
WITH   │ NO-ATTR-SPACE                                                │
       │ NO-BOX                                                       │
       │ NO-HIDE                                                      │  ...
       │ NO-LABELS                                                    │
       │ NO-UNDERLINE                                                 │
       │ NO-VALIDATE                                                  │
       │ OVERLAY                                                      │
       │ PAGE-BOTTOM                                                  │
       │ PAGE-TOP                                                     │
       │ RETAIN n                                                     │
       │ ROW  expression                                              │
       │ SCROLL  n                                                    │
       │ SIDE-LABELS                                                  │
       │ TITLE [ COLOR  color-phrase     ] expression                 │
       │ TOP-ONLY                                                     │
       └ WIDTH  n                                                     ┘
```

For more information on *frame-phrase*, see the Frame Phrase reference page.

EDITING-*phrase*

Identifies processing to take place as each keystroke is entered.

Here is the syntax of EDITING-*phrase*:

```
[ label :] EDITING: statement...    END.
```

For more information on EDITING-*phrase*, see the EDITING Phrase reference page.

record

The name of a record buffer. All of the fields in the record are processed exactly as if you had updated each of them individually.

To update a record in a file defined for multiple databases, you must qualify the record's filename with the database name. See the description of the Record Phrase for more information.

EXCEPT *field*

The fields in the EXCEPT phrase will be omitted from the update list.

EXAMPLES

r-updat.p

```
FOR EACH customer:
  UPDATE name address city st.
END.
```

The r-updat.p procedure lets you update the name, address, city, and state for each customer record in the database.

r-updat2.p

```
FOR EACH customer:
  UPDATE customer.name max-credit
    VALIDATE(max-credit < 10000, "Too high")
    HELP "enter max-credit < 10000".
  FOR EACH order OF customer:
    DISPLAY order-num.
    UPDATE pdate sdate
      VALIDATE(sdate > TODAY,
        "Ship date must be later than today").
  END.
END.
```

The r-updat2.p procedure reads each customer record and lets you update the name and max-credit fields. The VALIDATE option on the first UPDATE statement ensures that you enter a max-credit value that is less than 10000. The HELP option displays a help message to that effect.

The second FOR EACH block reads every order belonging to the customer, displays the order-num field and lets you update the pdate (promise date) and sdate (ship date) fields. The VALIDATE option ensures that you enter a ship date value that is after today's date.

r-updat3.p

```
REPEAT:
  PROMPT-FOR customer.cust-num.
  FIND customer USING cust-num.
  UPDATE name address city customer.st
    WITH 1 COLUMN 1 DOWN.
END.
```

This procedure requests a customer number and then lets you update information for the customer record with that number. The frame phrase "WITH 1 COLUMN 1 DOWN" tells PROGRESS to display the fields in a single column on the screen (rather than in a row across the screen) and to display only one customer record on the screen at a time.

NOTES

- If an error occurs during the UPDATE statement (e.g. the user enters a duplicate index value for a unique index), PROGRESS retries the data entry part of the statement, and does not do the error processing associated with the block containing the statement.
- Because the UPDATE statement is a combination of the DISPLAY, PROMPT-FOR, and ASSIGN statements, and the SET statement is a combination of the PROMPT-FOR and ASSIGN statements, it may seem that the UPDATE statement is equivalent to a combination of the DISPLAY and SET statements. However, this is not true. For example:

```
REPEAT:
   PROMPT-FOR customer.cust-num.
   FIND customer USING cust-num.
   UPDATE max-credit.
END.
```

This procedure is approximately equivalent to the following procedure:

```
REPEAT:
   PROMPT-FOR customer.cust-num.
   FIND customer USING cust-num.
   DISPLAY max-credit.
   DO ON ERROR UNDO, RETRY:
      SET max-credit.
   END.
END.
```

If an error occurs during an UPDATE statement, the statement is retried until the error is corrected. If this happens during a SET statement, an entire block is retried.

- If you are getting input from a device other than the terminal, and the number of characters read by the UPDATE statement for a particular field or variable exceeds the display format for that field or variable, PROGRESS returns an error. However, if you are setting a logical field that has a format of "y/n" and the data file contains a value of "yes" or "no", PROGRESS converts that value to "y" or "n".

- If you use a single qualified identifier with the UPDATE statement, the compiler first interprets the reference as *dbname.filename*. If the compiler cannot resolve the reference as *dbname.filename*, it tries to resolve it as *filename.fieldname*.

- When updating fields, you must use filenames that are different from field names to avoid ambiguous references. See the description of the Record Phrase for more information.

SEE ALSO ASSIGN Statement, DISPLAY Statement, EDITING Statement, Format Phrase, Frame Phrase, PROMPT-FOR Statement

UPDATE Statement (SQL)

Changes values in one or more rows of a table.

SYNTAX

UPDATE *table-name*
    SET *column-name* = *expression* [, *column-name* = *expression*] ...
        [WHERE { *search-condition* | { CURRENT OF *cursor-name* }}]

table-name
The name of the table in which you want to update rows.

SET *column-name* = *expression* [, *column-name* = *expression*,...]
Assigns new values to each column named. The expression can be the keyword NULL, or it can be a literal, column name, arithmetic operation, a defined PROGRESS variable or field name, or any combination of these. The column name may not be qualified.

WHERE *search-condition*
Identifies the conditions under which the rows are updated. The search condition compares the values in one column to the values in another column or to a literal. If you omit the WHERE clause, all rows of the target table are updated. For more detailed information about WHERE clause search conditions, see Chapter 15 of the *Programming Handbook.*

WHERE CURRENT OF *cursor-name*
Identifies the cursor that points to the row to be updated (positioned update).

EXAMPLE

```
UPDATE item
        SET cost = cost * 1.25
        WHERE item-num = 52.
```

NOTE

- The UPDATE statement can be used in both interactive SQL and embedded SQL.

USERID Function

Returns the userid of the current user.

SYNTAX

```
USERID [ ( logical-dbname ) ]
```

logical-dbname

The logical name of the database from where you want to retrieve your user ID. The logical database name must be a character string enclosed in quotes, or a character expression. If you do not specify this argument, the compiler inserts the name of the database that is connected when the procedure is compiled. If you omit this argument and more than one database is connected, a compiler error results.

EXAMPLE

r-userid.p

```
DISPLAY USERID LABEL "You are logged in as"
            WITH SIDE-LABELS.
```

This one line procedure displays your current userid.

NOTES

- When using the USERID function, you receive a compiler error under the following conditions:
  - There is no database connected.
  - The *logical-dbname* argument is omitted, and more than one database is currently connected.
- When specifying the *logical-dbname* argument, you must provide the name of the logical database, not the physical database.
- Every user who enters PROGRESS is given an initial userid. Tables 31 through 34 show how PROGRESS determines a user's initial userid.

Table 31: Determining a UNIX Userid

Are there records in the _User file?	Were the -U and -P start-up options supplied?	Userid
NO	YES	Error: -U and -P not allowed unless there areentries in the _User file.
NO	NO	UNIX userid
YES	NO	“” (blank userid)
YES	YES	If the -U useridand -P password match those in the _User file, use that userid. Otherwise, do not assign a userid.

Table 32: Determining a DOS or OS/2 Userid

Are there records in the _User file?	Were the -U and -P start-up options supplied?	Userid
NO	YES	Error: -U and -P not allowed unless there areentries in the _User file.
NO	NO	”” (blank userid)
YES	NO	“” (blank userid)
YES	YES	If the -U userid and -P password match those in the _User file,use that userid. Otherwise, do not assign a userid.

Table 33: Determining a VMS Userid

Are there records in the _User file?	Were the /USER and /PASSWORD options supplied?	Userid
NO	YES	Error: These options are not allowed unless there are entries in the _User file.
NO	NO	VMS userid
YES	NO	"" (blank userid)
YES	YES	If the /USERID and /PASSWORD match those in the _User file, use that userid. Otherwise, do not assign a userid.

Table 34: Determining a BTOS/CTOS Userid

Are there records in the _User file?	Were the -U and -P start-up options supplied?	Userid
NO	YES	Error: -U and -P not allowed unless there are entries in the _User file.
NO	NO	BTOS/CTOS user name
YES	NO	"" (blank userid)
YES	YES	If the BTOS/CTOS userid and password parameters match those in the _User file, use that userid. Otherwise, do not assign a userid.

- After PROGRESS starts running, you can use the SETUSERID function to change the current userid.

SEE ALSO CONNECT Statement, SETUSERID Function, Chapter 11 of the *Programming Handbook*

VALIDATE Statement

Verifies that a record complies with mandatory field and unique index definitions.

SYNTAX

```
VALIDATE record
```

record

The name of the record you want to validate.

To validate a record in a file defined for multiple databases, you must qualify the record's filename with the database name. See the description of the Record Phrase for more information.

EXAMPLE

r-valid.p

```
REPEAT FOR item:
  PROMPT-FOR item-num.
  FIND item USING item-num NO-ERROR.
  IF NOT AVAILABLE item
  THEN DO:
    CREATE item.
    ASSIGN item-num.
    UPDATE idesc cost subs-item.
    VALIDATE item.
  END.
  ELSE DISPLAY idesc cost subs-item.
END.
```

In this example, the procedure prompts for an item number. If an item with that number is not available, the procedure creates a new item record and lets you supply some item information. The VALIDATE statement checks the data you enter against the index and mandatory field criteria for the item record.

NOTES

- Because validation is done automatically, you rarely need to use the VALIDATE statement explicitly. PROGRESS automatically validates a record whenever a record in the record buffer is replaced by another, the record's scope iterates or ends, the innermost iterating subtransaction block which creates a record iterates, or at the end of a transaction. For more information on record scope, see Chapter 5 of the *Programming Handbook*. For more information on subtransaction blocks, see Chapter 8 of the *Programming Handbook*.

- PROGRESS automatically validates mandatory fields when those fields are modified.
- If the validation fails on a newly-created record, VALIDATE raises the ERROR condition.
- PROGRESS performs validation when it leaves a field.
- For complex validations, it may be easier to use the IF...THEN...ELSE statement instead of the VALIDATE statement.
- You cannot use the VALIDATE statement to test fields that are referenced in SQL statements, since validation is not performed for these fields.

SEE ALSO IF...THEN...ELSE Statement

VIEW Statement

Brings a frame into view, or activates the frame for display at the beginning or end of a page (if it is a PAGE-TOP or PAGE-BOTTOM frame).

SYNTAX

```
VIEW [ STREAM stream ]  [ FRAME frame ]
```

STREAM *stream*

Specifies the name of a stream. If you do not name a stream, PROGRESS uses the unnamed stream. See the DEFINE STREAM reference page and Chapter 9 of the *Programming Handbook* for more information about streams.

FRAME *frame*

The name of the frame you want to view. If you do not use this option, VIEW displays the default frame for the current block.

EXAMPLES

r-view.p

```
FORM "Please choose one of:" SKIP(1)
     "1 - order entry"       SKIP
     "2 - invoices    "      SKIP
     "3 - exit        "
     WITH CENTERED FRAME menu.

REPEAT:
  VIEW FRAME menu.
  READKEY.
  IF LASTKEY = KEYCODE("1") THEN RUN ordentry.
  ELSE
  IF LASTKEY = KEYCODE("2") THEN RUN invoice.
  ELSE
  IF LASTKEY = KEYCODE("3") THEN LEAVE.
  ELSE MESSAGE "Sorry, that is not in the list".
END.
```

The r-view.p procedure uses a FORM statement to describe the menu frame. Because the FORM statement describes a frame but does not display it, the procedure uses a VIEW statement to bring the menu frame into view.

r-view2.p

```
OUTPUT TO slsrep PAGED PAGE-SIZE 10.
FOR EACH salesrep:
  PAGE.
  FORM HEADER "Sales rep report" "Page"
    AT 60 PAGE-NUMBER FORMAT ">>>9".
  DISPLAY SKIP(1) sales-rep slsname slsrgn
    WITH NO-LABELS.

  FORM HEADER "Sales rep report" sales-rep
    "(continued)" "Page" AT 60 PAGE-NUMBER.
    FORMAT ">>>9" SKIP(1)
    WITH FRAME hdr2 PAGE-TOP.
  VIEW FRAME hdr2.

  FOR EACH customer OF salesrep:
    DISPLAY cust-num name address city st.
  END.
END.
```

The r-view2.p procedure displays information about a salesrep and then displays all the customers belonging to that salesrep. Each time there is a new salesrep, the report begins a new page. In addition, if the information for a salesrep takes up more than one page, a separate FORM statement describes a continuation header for that salesrep. The VIEW statement for the PAGE-TOP frame hdr2 causes it to become "active" for subsequent page breaks, but does not cause it to be output immediately.

NOTES

- If the frame is already displayed, the VIEW statement has no effect.
- If there is not enough room on the screen for the frame, PROGRESS removes other frames, starting from the bottom of the screen, until there is room for the new frame.

SEE ALSO DEFINE STREAM Statement, HIDE Statement

VMS Statement

Runs a program, VMS command or VMS command file, or starts an interactive VMS command processor.

SYNTAX

```
                            ┌                        ┌                      ┐    ┐
VMS [ SILENT] [ATTACH]      │ vms-command            │ argument             │... │
                            │ VALUE( expression )    │ VALUE(expression  )  │    │
                            └                        └                      ┘    ┘
```

SILENT

After processing a VMS statement, PROGRESS pauses and asks you to press the space bar to continue. You can use the SILENT option to eliminate this pause. Use this option only if you are sure that the VMS command does not generate any output to the screen.

ATTACH

Attach to a pre-existing VMS process. By attaching to a pre-existing process, you avoid the overhead of restarting the process each time you want to do work in the VMS environment.

vms-command

The name of the program, command, or batch file you want to run. If you do not use this option, the VMS Statement invokes the VMS shell and remains there until you type LOGOFF.

VALUE*(expression)*

expression is a constant, field name, variable name, or any combination of these, whose value is the name of a program, command, or batch file you want the VMS statement to use.

argument

One or more literal arguments you want to pass to the program, command, or batch file being run.

VALUE*(expression)*

expression is a constant, field name, variable name, or any combination of these, whose value is an argument you want to pass to the program, command or batch file being run.

EXAMPLE

r-vms.p

```
DEFINE VARIABLE proc AS CHARACTER
  FORMAT "x(10)".

REPEAT:
  DISPLAY "Enter L to list your files"
    WITH ROW 5 CENTERED FRAME a.
  SET proc LABEL "Enter a valid Procedure Name"
    WITH ROW 9 CENTERED FRAME b.
  IF proc = "L" THEN
    IF OPSYS = "vms" THEN VMS directory.
    ELSE IF OPSYS = "unix" THEN UNIX ls.
    ELSE IF OPSYS = "msdos" THEN DOS dir.
    ELSE IF OPSYS = "os2" then OS2 dir.
    ELSE BTOS "[sys]<sys>files.run" files.
  ELSE DO:
    HIDE FRAME a.
    HIDE FRAME b.
    RUN VALUE(proc).
  END.
END.
```

Here, if you enter an "L", PROGRESS runs either the VMS dir, UNIX ls, DOS dir command, OS2 dir command, or BTOS [sys]<sys>files.run files. If you enter a procedure name, that name is stored in the proc variable. The RUN statement runs the value of the proc variable.

NOTES

- If you are using DOS, OS/2, BTOS/CTOS, or UNIX and you use the VMS statement in a procedure, that procedure will compile. The procedure will run as long as flow of control does not pass through the VMS statement.

- This command does not exit to VMS and return. It spawns a VMS command processor within PROGRESS to execute the command. Because of this, you cannot use the VMS statement as a substitute for QUIT.

- You can also access an interactive VMS command processor by selecting option "e" from the main menu of PROGRESS Help.

SEE ALSO DOS Statement, OPSYS Function, UNIX Statement, BTOS Statement, CTOS Statement, OS2 Statement.

WEEKDAY Function

Evaluates a date expression and returns the day of the week as an integer from 1 (Sunday) to 7 (Saturday) for that date.

SYNTAX

```
WEEKDAY(date)
```

date

A date expression (a constant, field name, variable name, or any combination of these that results in a date value) for which you want the day of the week.

EXAMPLE

r-wkday.p

```
DEFINE VARIABLE birth-date AS DATE
  LABEL "Birth Date".
DEFINE VARIABLE daynum AS INTEGER.
DEFINE VARIABLE daylist AS CHARACTER
  FORMAT "x(9)"
  INITIAL "Sunday,Monday,Tuesday,Wednesday,
           Thursday,Friday,Saturday".
DEFINE VARIABLE dayname AS CHARACTER
  LABEL "Day You Were Born".
DEFINE VARIABLE daysold AS INTEGER
  LABEL "Days Since You Were Born".

REPEAT:
  SET birth-date.
  daynum = WEEKDAY(birth-date).
  dayname = ENTRY(daynum,daylist).
  daysold = TODAY - birth-date.
  DISPLAY dayname daysold.
END.
```

This procedure tells you the day of the week on which you were born and how many days old you are.

SEE ALSO DAY Function, MONTH Function, YEAR Function

YEAR Function

Evaluates a date expression and returns the year value of that date, including the century.

SYNTAX

```
YEAR(date )
```

date

A date expression (a constant, field name, variable name, or any combination of these that results in a date value) for which you want to determine the year.

EXAMPLE

r-year.p

```
DEFINE VARIABLE outfmt AS CHARACTER.
DEFINE VARIABLE orddate AS CHARACTER
  LABEL "Order Date" FORMAT "x(10)".

FOR EACH order:
  IF YEAR(odate) >= 2000
  THEN outfmt = "99/99/9999".
  ELSE outfmt = "99/99/99".
  orddate = STRING(odate,outfmt).
  DISPLAY order.order-num orddate terms.
END.
```

This procedure uses the YEAR function to determine whether an order date is in this century or the next, and then uses a different display format for each.

SEE ALSO DAY Function, MONTH Function, WEEKDAY Function

Keyword Index

The keyword index has three columns:

- *Keyword* shows the keyword or language component in its normal form.
- *Minimum Abbreviation* shows the shortest allowable abbreviation of the keyword or component that PROGRESS will recognize.
- *Usage* is a brief description of the way the keyword is used. A keyword may have several different uses, depending on the context in which you use it. This book gives detailed information about each statement, phrase, function, and operator described in this column.

Keyword	**Minimum Abbreviation**	**Usage**
(	(	Expression begin delimiter
)	)	Expression end delimiter
*	*	Numeric operator
*/	*/	Comment end delimiter
+	+	Character operator Date operator Numeric operator
,	,	Separator (FOR EACH, UNDO statements; ON ERROR, ON ENDKEY phrases)
–	–	Date operator Numeric operator
.	.	Separator (qualified field names) Terminator (non-blank statement)
/	/	Numeric operator
/*	/*	Comment begin delimiter

Keyword	Minimum Abbreviation	Usage
:	:	Termination symbol (block statement)
;	;	Alternative representation character
		Termination symbol (for embedded SQL statements only)
<	<	Logical operator
< =	< =	Logical operator
< >	< >	Logical operator
=	=	Assignment operator
		Logical operator
>	>	Logical operator
> =	> =	Logical operator
?	?	Unknown value symbol
[	[	Array subscript begin delimiter
\	\	Escape character (UNIX only)
]	]	Array subscript end delimiter
{	{	Compile-time substitution end delimiter
}	}	Compile-time substitution begin delimiter
~	~	Escape character
@	@	Option (DISPLAY statement)
^	^	Option (PROMPT-FOR, SET, UPDATE statements)
ACCUM	ACCUM	Function
ACCUMULATE	ACCUM	Statement
ADD	ADD	SQL keyword
ALIAS	ALIAS	Reserved by Compiler.
ALL	ALL	Option (CLEAR, HIDE statements)
		SQL keyword
ALTER	ALTER	ALTER TABLE SQL Statement
AMBIGUOUS	AMBIG	Function
AND	AND	Logical operator
		Option (Record phrase)
		SQL keyword

Keyword	Minimum Abbreviation	Usage
ANY	ANY	SQL keyword
APPEND	APPEND	Option (OUTPUT statement)
APPLY	APPLY	Statement
AS	AS	Option (DEFINE PARAMETER, DEFINE VARIABLE, DEFINE WORKFILE statements) SQL keyword
ASC	ASC	Function SQL keyword
ASSIGN	ASSIGN	Statement
AT	AT	Option (Format phrase)
ATTACH	ATTACH	Option (VMS statement)
ATTR-SPACE	ATTR	Option (Format, Frame phrases; COMPILE, PUT SCREEN statements)
AUTHORIZATION	AUTHORIZATION	SQL keyword
AUTO-RETURN	AUTO-RET	Option (CHOOSE statement; Format phrase)
AVAILABLE	AVAIL	Function
AVERAGE	AVE	Option (Aggregate phrase)
AVG	AVG	SQL keyword
BEFORE-HIDE	BEFORE-HIDE	Option (PAUSE statement)
BEGINS	BEGINS	Function
BELL	BELL	Statement
BETWEEN	BETWEEN	SQL keyword
BLACK	BLA	Option (Color phrase)
BLANK	BLANK	Option (Format phrase)
BLINK	BLI	Option (Color phrase)
BLUE	BLU	Option (Color phrase)
BREAK	BREAK	Option (DO, FOR EACH, REPEAT statements)
BRIGHT	BRI	Option (Color phrase)
BROWN	BRO	Option (Color phrase)
BTOS	BTOS	Statement
BUFFER	BUFFER	Keyword (DEFINE BUFFER statement)
BY	BY	Option (DO, FOR EACH, REPEAT statement) SQL keyword

Keyword Index

Keyword	Minimum Abbreviation	Usage
CALL	CALL	Statement
CAN-DO	CAN-DO	Function
CAN-FIND	CAN-FIND	Function
CAPS	CAPS	Function
CASE-SENSITIVE	CASE	Option (ALTER TABLE, CREATE TABLE DEFINE VARIABLE, DEFINE WORKFILE, Statements)
CENTERED	CENTER	Option (Frame phrase)
CHAR	CHAR	abbreviation
CHARACTER	C	Option (DEFINE PARAMETER, DEFINE VARIABLE, DEFINE WORKFILE statements; Format phrase) SQL keyword
CHECK	CHECK	SQL keyword
CHOOSE	CHOOSE	Statement
CHR	CHR	Function
CLEAR	CLEAR	Option (INPUT statement)
CLOSE	CLOSE	Keyword (INPUT CLOSE, INPUT-OUTPUT CLOSE, OUTPUT CLOSE statements) Close Cursor SQL Statement
COL	COL	abbreviation
COLON	COLON	Option (Format phrase)
COLOR	COLOR	Statement Option (Frame phrase; CHOOSE, MESSAGE, PUT SCREEN statements)
COLUMN	COL	Option (Frame phrase; PUT CURSOR, PUT SCREEN statements) SQL keyword
COLUMN-LABEL	COLUMN-LABEL	Option (DEFINE PARAMETER, DEFINE VARIABLE, DEFINE WORKFILE statements; Format phrase)
COLUMNS	COLUMNS	Option (Frame phrase)
COMMIT	COMMIT	COMMIT WORK SQLStatement
COMPILE	COMPILE	Statement
CONNECT	CONNECT	Statement
CONNECTED	CONNECTED	Function

Keyword	Minimum Abbreviation	Usage
CONTROL	CONTROL	Option (PUT statement)
COUNT	COUNT	Option (Aggregate phrase) SQL keyword
COUNT-OF	COUNT-OF	Function
CREATE	CREATE	Statement CREATE ALIAS, CREATE INDEX, CREATE TABLE, CREATE VIEW SQL Statements
CTOS	CTOS	Statement
CURRENT	CURRENT	SQL keyword
CURSOR	CURSOR	Keyword (PUT CURSOR statement), SQL keyword (DECLARE CURSOR statement)
CYAN	C	Option (Color phrase)
DARK-GRAY	D-GRAY	Option (Color phrase)
DATABASE	DATABASE	Reserved for compiler.
DATE	DA	Option (DEFINE PARAMETER, DEFINE VARIABLE statement; Format phrase)
DATE	DATE	Function
DAY	DAY	Function
DBNAME	DBNAME	Function
DBRESTRICTIONS	DBREST	Function
DBTYPE	DBTYPE	Function
DBVERSION	DBVERS	Function
DEBLANK	DEBLANK	Option (Format phrase)
DEC	DE	Abbreviation
DECIMAL	DEC	Function SQL keyword
DECIMAL	DE	Option (DEFINE PARAMETER, DEFINE VARIABLE, DEFINE WORKFILE statements; Format phrase)
DECIMALS	DECIMALS	Option (DEFINE PARAMETER, DEFINE VARIABLE, DEFINE WORKFILE statements)
DECLARE	DECLARE	DECLARE CURSOR SQL Statement
DEF	DEF	Abbreviation

Keyword	Minimum Abbreviation	Usage
DEFAULT	DEFAULT	Option (STATUS statement)
DEFINE	DEF	Keyword (DEFINE BUFFER, DEFINE PARAMETER, DEFINE SHARED FRAME, DEFINE STREAM, DEFINE VARIABLE, DEFINE WORKFILE statements)
DELETE	DELETE	Statement SQL keyword (DELETE FROM statement)
DESC	DESC	SQL keyword
DESCENDING	DESC	Option (DO, FOR EACH, REPEAT statements)
DICTIONARY	DICT	Statement
DISCONNECT	DISCON	Statement
DISPLAY	DISP	Statement Option (COLOR statement)
DISTINCT	DISTINCT	SQL keyword
DO	DO	Statement
DOS	DOS	Statement
DOUBLE	DOUBLE	SQL keyword
DOWN	DOWN	Statement Option (Frame phrase; SCROLL statement)
DROP	DROP	SQL keyword (DROP INDEX, DROP TABLE, DROP VIEW statement)
EACH	EACH	Keyword (FOR EACH statement) Option (Record phrase)
ECHO	ECHO	Option (INPUT FROM, INPUT THROUGH, INPUT-OUTPUT THROUGH, OUTPUT THROUGH statements)
EDITING	EDITING	Phrase (PROMPT-FOR, SET, UPDATE statements)
ELSE	ELSE	Keyword (IF...THEN...ELSE statement; IF...THEN...ELSE function)
ENCODE	ENCODE	Statement
END	END	Statement SQL keyword
ENDKEY	ENDKEY	Keyword (ON ENDKEY phrase)
ENTERED	ENTERED	Function
ENTRY	ENTRY	Function

Keyword	Minimum Abbreviation	Usage
EQ	EQ	Logical operator
ERROR	ERROR	Keyword (ON ERROR phrases)
ESCAPE	ESCAPE	SQL keyword
EXCEPT	EXCEPT	Option (ASSIGN, DISPLAY, EXPORT, FORM, IMPORT, INSERT, PROMPT-FOR, SET, UPDATE statements)
EXCLUSIVE-LOCK	EXCLUSIVE	Option (Record phrase)
EXISTS	EXISTS	SQL keyword
EXP	EXP	Function
EXPORT	EXPORT	Statement
EXTENT	EXTENT	Option (DEFINE VARIABLE, DEFINE WORKFILE statements)
FALSE	FALSE	Logical value
FETCH	FETCH	SQL Statement
FIELD	FIELD	Option (CHOOSE, DEFINE WORKFILE statements)
FILL	FILL	Function
FIND	FIND	Statement
FIRST	FIRST	Function Option (FIND statement)
FIRST-OF	FIRST-OF	Function
FLOAT	FLOAT	SQL keyword
FOR	FOR	Keyword (DEFINE BUFFER, DO, FOR EACH, REPEAT statements) SQL keyword
FORM	FORM	Statement
FORMAT	FORMAT	Option (DEFINE PARAMETER, DEFINE VARIABLE, DEFINE WORKFILE statements; Format phrase)
FRAME	FRAME	Option (CLEAR statement; Frame phrase) Keyword (DEFINE SHARED FRAME statement)
FRAME-DB	FRAME-DB	Function
FRAME-COL	FRAME-COL	Function
FRAME-DOWN	FRAME-DOWN	Function

Keyword	Minimum Abbreviation	Usage
FRAME-FIELD	FRAME-FIELD	Function
FRAME-FILE	FRAME-FILE	Function
FRAME-INDEX	FRAME-INDEX	Function
FRAME-LINE	FRAME-LINE	Function
FRAME-NAME	FRAME-NAME	Function
FRAME-ROW	FRAME-ROW	Function
FRAME-VALUE	FRAME-VAL	Function Statement
FROM	FROM	Keyword (INPUT statement) SQL keyword
FROM-CURRENT	FROM-CURRENT	Option (SCROLL statement)
GATEWAYS	GATEWAYS	Function
GE	GE	Logical operator
GETBYTE	GETBYTE	Statement
GLOBAL	GLOBAL	Option (DEFINE STREAM, DEFINE VARIABLE statements)
GO-ON	GO-ON	Option (CHOOSE, PROMPT-FOR, SET, UPDATE statements)
GO-PENDING	GO-PEND	Function
GRANT	GRANT	SQL Statement
GRAY	GRA	Option (Color phrase)
GREEN	GRE	Option (Color phrase)
GROUP	GROUP	SQL keyword
GT	GT	Logical operator
HAVING	HAVING	SQL keyword
HEADER	HEADER	Option (FORM statement)
HELP	HELP	Option (Format phrase)
HIDE	HIDE	Statement
IF	IF	Keyword (IF...THEN...ELSE function; IF...THEN...ELSE statement)
IMPORT	IMPORT	Statement
IN	IN	SQL keyword

INDEX	INDEX	Function SQL keyword
INDICATOR	INDICATOR	SQL keyword
INITIAL	INIT	Option (DEFINE PARAMETER, DEFINE VARIABLE, DEFINE WORKFILE statements)
INPUT	INPUT	Function Keyword (DEFINE PARAMETER, INPUT CLEAR, INPUT CLOSE, INPUT FROM, INPUT THROUGH, INPUT-OUTPUT CLOSE, INPUT-OUTPUT THROUGH statements) Option (Color phrase, RUN statement)
INPUT-OUTPUT	INPUT-OUTPUT	Statement (not on DOS) Keyword (DEFINE statement)
INSERT	INSERT	Statement SQL keyword
INT	INT	Abbreviation
INTEGER	I	Option (DEFINE VARIABLE, DEFINE WORKFILE statements; Format phrase) SQL keyword
INTEGER	INT	Function
INTO	INTO	SQL keyword
IS	IS	SQL keyword
IS-ATTR-SPACE	IS-ATTR	Function
KBLABEL	KBLABEL	Function
KEYCODE	KEYCODE	Function
KEYFUNCTION	KEYFUNC	Function
KEYLABEL	KEYLABEL	Function
KEYS	KEYS	Option (CHOOSE statement)
KEYWORD	KEYWORD	Function
LABEL	LABEL	Option (DEFINE PARAMETER, DEFINE VARIABLE, DEFINE WORKFILE statements; Format phrase)
LAST	LAST	Function Option (FIND statement)
LAST-OF	LAST-OF	Function
LASTKEY	LASTKEY	Function
LDBNAME	LDBNAME	Function
LC	LC	Function

Keyword	Minimum Abbreviation	Usage
LE	LE	Logical operator
LEAVE	LEAVE	Statement Option (ON ENDKEY, ON ERROR phrases; UNDO statement)
LENGTH	LENGTH	Function, Statement
LIBRARY	LIBRARY	Function
LIGHT	LI	Option (Color phrase)
LIKE	LIKE	Option (DEFINE VARIABLE, DEFINE WORKFILE statements; Format phrase) SQL keyword
LINE-COUNTER	LINE-COUNT	Function
LISTING	LISTING	Option (COMPILE statement)
LOCKED	LOCKED	Function
LOG	LOG	Function
LOGICAL	L	Option (DEFINE VARIABLE statement; Format phrase)
LOOKUP	LOOKUP	Function
LT	LT	Logical operator
MAGENTA	MA	Option (Color phrase)
MAP	MAP	Option (INPUT FROM, INPUT THROUGH, INPUT-OUTPUT THROUGH, OUTPUT THROUGH, OUTPUT TO statements)
MATCHES	MATCHES	Function
MAX	MAX	Abbreviation SQL keyword
MAXIMUM	MAX	Option (Aggregate phrase)
MEMBER	MEMBER	Function
MESSAGE	MESSAGE	Statement Option (HIDE statement)
MESSAGES	MESSAGES	Option (Color phrase)
MESSAGE-LINES	MESSAGE-LINES	Function
MIN	MIN	Abbreviation SQL keyword
MINIMUM	MIN	Option (Aggregate phrase)

Keyword	Minimum Abbreviation	Usage
MODULO	MOD	Function
MONTH	MONTH	Function
NE	NE	Logical operator
NEW	NEW	Option (DEFINE BUFFER, DEFINE SHARED FRAME, DEFINE STREAM, DEFINE VARIABLE, DEFINE WORKFILE statements)
NEXT	NEXT	Statement Option (FIND, UNDO statements)
NEXT-PROMPT	NEXT-PROMPT	Statement
NO	NO	Logical value
NO-ATTR-SPACE	NO-ATTR	Option (Format, Frame phrases; COMPILE, PUT SCREEN statements)
NO-BOX	NO-BOX	Option (Frame phrase)
NO-ECHO	NO-ECHO	Option (INPUT FROM, INPUT THROUGH, INPUT-OUTPUT THROUGH, OUTPUT THROUGH statements)
NO-ERROR	NO-ERROR	Option (CHOOSE, FIND statements)
NO-HIDE	NO-HIDE	Option (Frame phrase)
NO-LABEL	NO-LABEL	Option (Format phrase)
NO-LABELS	NO-LABELS	Option (Frame phrase)
NO-LOCK	NO-LOCK	Option (Record phrase)
NO-MAP	NO-MAP	Option (INPUT FROM, INPUT THROUGH, INPUT-OUTPUT THROUGH, OUTPUT THROUGH, OUTPUT TO statements)
NO-MESSAGE	NO-MESSAGE	Option (PAUSE statement)
NO-PAUSE	NO-PAUSE	Option (CLEAR, HIDE statements)
NO-UNDERLINE	NO-UNDERLINE	Option (Frame phrase)
NO-UNDO	NO-UNDO	Option (DEFINE VARIABLE, DEFINE WORKFILE statements)
NO-VALIDATE	NO-VAL	Option (Frame phrase)
NO-WAIT	NO-WAIT	Option (FIND statement)
NORMAL	NORMAL	Option (Color phrase)
NOT	NOT	Logical operator Function SQL keyword

Keyword	Minimum Abbreviation	Usage
NULL	NULL	Keyword (PUT statement) SQL keyword
NUM-ALIASES	NUM-ALIASES	Function
NUM-DBS	NUM-DBS	Function
NUM-ENTRIES	NUM-ENTRIES	Function
NUMERIC	NUMERIC	SQL keyword
OF	OF	Option (Record phrase) SQL keyword
OFF	OFF	Option (PUT CURSOR, STATUS statements)
ON	ON	Statement SQL keyword
OPEN	OPEN	OPEN CURSOR SQL Statement
OPSYS	OPSYS	Function
OPTION	OPTION	Reserved by Compiler.
OR	OR	Logical operator SQL keyword
ORDER	ORDER	SQL keyword
OS2	OS2	Statement
OUTPUT	OUTPUT	Keyword (DEFINE, OUTPUT CLOSE, OUTPUT THROUGH, OUTPUT TO statements) Option (RUN statement)
OVERLAY	OVERLAY	Function Option (Frame phrase)
PAGE	PAGE	Statement
PAGE-BOTTOM	PAGE-BOT	Option (Frame phrase)
PAGE-NUMBER	PAGE-NUM	Function
PAGE-SIZE	PAGE-SIZE	Function
PAGE-TOP	PAGE-TOP	Option (Frame phrase)
PAGE-WIDTH	PAGE-WIDTH	Option (COMPILE statement)
PAGED	PAGED	Option (OUTPUT THROUGH, OUTPUT TO statements)
PARAMETER	PARAM	Keyword (DEFINE PARAMETER statement)
PAUSE	PAUSE	Statement Option (CHOOSE statement)

Keyword	Minimum Abbreviation	Usage
PDBNAME	PDBNAME	Function
PRECISION	PRECISION	SQL keyword
PRESELECT	PRESEL	Option (DEFINE BUFFER, DO, REPEAT statements)
PREV	PREV	Option (FIND statement)
PRINTER	PRINTER	Option (OUTPUT TO statement)
PRIVILEGES	PRIVILEGES	SQL keyword
PROGRAM-NAME		PROGRAM-NAME Function
PROGRESS	PROGRESS	Function
PROMPT	PROMPT	Option (COLOR statement)
PROMPT-FOR	PROMPT	Statement
PROPATH	PROPATH	Statement Function
PUBLIC	PUBLIC	SQL keyword
PUT	PUT	Statement Keyword (PUT CURSOR, PUT SCREEN statements)
PUTBYTE	PUTBYTE	Statement
QUIT	QUIT	Statement
R-INDEX	R-INDEX	Function
RANDOM	RANDOM	Function
RAW	RAW	Statement, Function
READKEY	READKEY	Statement
REAL	REAL	SQL keyword
RECID	R	Option (DEFINE VARIABLE statement; Format phrases)
RECID	RECID	Function
RED	RED	Option (Color phrase)
RELEASE	RELEASE	Statement
REPEAT	REPEAT	Statement
RETAIN	RETAIN	Option (Frame phrase)
RETRY	RETRY	Function Option (UNDO statement)

Keyword	Minimum Abbreviation	Usage
RETURN	RETURN	Statement Option (UNDO statement)
REVERSE	REVERSE	Option (Color phrase)
REVOKE	REVOKE	REVOKE PRIVILEGES SQL Statement
ROLLBACK	ROLLBACK	ROLLBACK WORK SQL Statement
ROUND	ROUND	Function
ROW	ROW	Option (Frame phrase; CHOOSE, PUT CURSOR, PUT SCREEN statements)
RUN	RUN	Statement
SAVE	SAVE	Option (COMPILE statement)
SCHEMA	SCHEMA	SQL keyword
SCREEN	SCREEN	Keyword (PUT SCREEN statement)
SCREEN-LINES	SCREEN-LINES	Function
SCROLL	SCROLL	Statement Option (Frame phrase)
SDBNAME	SDBNAME	Function
SEARCH	SEARCH	Function
SEEK	SEEK	Function Statement
SELECT	SELECT	SQL keyword
SET	SET	Statement Option (MESSAGE statement) SQL keyword
SETUSERID	SETUSERID	Function
SHARE-LOCK	SHARE-LOCK	Option (Record phrase)
SHARED	SHARED	Option (DEFINE BUFFER, DEFINE FRAME, DEFINE STREAM, DEFINE VARIABLE, DEFINE WORKFILE statements)
SIDE-LABELS	SIDE-LABELS	Option (Frame phrase)
SILENT	SILENT	Option (BTOS, CTOS, DOS, OS2, UNIX, VMS statements)
SKIP	SKIP	Option (DISPLAY, PROMPT-FOR, PUT,SET, UPDATE statements)
SMALLINT	SMALLINT	SQL keyword

Keyword	Minimum Abbreviation	Usage
SOME	SOME	SQL keyword
SPACE	SPACE	Option (DISPLAY, PROMPT-FOR, PUT,SET, UPDATE statements)
SQRT	SQRT	Function
STATUS	STATUS	Statement
STOP	STOP	Statement
STREAM	STREAM	Keyword (DEFINE STREAM statement) Option (DISPLAY, DOWN, EXPORT, HIDE, INPUT CLOSE, INPUT FROM, INPUT THROUGH, INPUT-OUTPUT CLOSE, INPUT-OUTPUT THROUGH, OUTPUT CLOSE, OUTPUT THROUGH, OUTPUT TO, PAGE, PROMPT-FOR, PUT, READKEY, UNDERLINE, UP, VIEW statements)
STRING	STRING	Function
SUB-AVERAGE	SUB-AVE	Option (Aggregate phrase)
SUB-COUNT	SUB-COUNT	Option (Aggregate phrase)
SUB-MAXIMUM	SUB-MAX	Option (Aggregate phrase)
SUB-MINIMUM	SUB-MIN	Option (Aggregate phrase)
SUB-TOTAL	SUB-TOTAL	Option (Aggregate phrase)
SUBSTRING	SUBSTR	Function
SUM	SUM	SQL keyword
TABLE	TABLE	SQL keyword
TERMINAL	TERMINAL	Option (INPUT FROM, OUTPUT TO statements)
TERMINAL	TERM	Function
TEXT	TEXT	Option (PROMPT-FOR, SET and UPDATE statements)
THEN	THEN	Keyword (IF...THEN...ELSE function; IF...THEN...ELSE statement)
THROUGH	THROUGH	Keyword (INPUT THROUGH, INPUT-OUTPUT THROUGH, OUTPUT THROUGH statements)
THRU	THRU	Keyword (INPUT THROUGH, INPUT-OUTPUT THROUGH, OUTPUT THROUGH statements)

Keyword	Minimum Abbreviation	Usage
TIME	TIME	Function
TITLE	TITLE	Option (FORM statement; Frame phrase)
TODAY	TODAY	Function
TO	TO	Keyword (OUTPUT TO statement) Option (Format phrase; DO, REPEAT statements) SQL keyword
TOP-ONLY	TOP-ONLY	Option (Frame phrase)
TOTAL	TOTAL	Option (Aggregate phrase)
TRANSACTION	TRANS	Option (DO, FOR EACH, REPEAT statements)
TRIM	TRIM	Function
TRUE	TRUE	Logical value
TRUNCATE	TRUNC	Function
UNBUFFERED	UNBUFF	Option (INPUT FROM, INPUT THROUGH, INPUT-OUTPUT THROUGH, OUTPUT THROUGH, OUTPUT TO statements)
UNDERLINE	UNDERL	Statement Option (Color phrase)
UNDO	UNDO	Statement Option (ON ENDKEY, ON ERROR phrases)
UNFORMATTED	UNFORMATTED	Option (PUT statement)
UNION	UNION	SQL keyword
UNIQUE	UNIQUE	Reserved by Compiler SQL keyword
UNIX	UNIX	Statement
UP	UP	Statement Option (SCROLL statement)
UPDATE	UPDATE	Statement Option (MESSAGE statement) SQL keyword
USE-INDEX	USE-INDEX	Option (Record phrase)
USER	USER	SQL keyword
USERID	USERID	Function
USING	USING	Option (Record phrase)
VALIDATE	VALIDATE	Statement Option (Format phrase)

Keyword	Minimum Abbreviation	Usage
VALUE	VALUE	Option (BTOS, CTOS, DOS, INPUT FROM, INPUT THROUGH, INPUT-OUTPUT THROUGH, OS2, OUTPUT THROUGH, RUN, UNIX, VMS statements)
VALUES	VALUES	SQL keyword
VARIABLE	VAR	Keyword (DEFINE VARIABLE statement)
VIEW	VIEW	Statement SQL keyword
VMS	VMS	Statement
WEEKDAY	WEEKDAY	Function
WHEN	WHEN	Option (DISPLAY, PROMPT-FOR, SET, UPDATE statements)
WHERE	WHERE	Option (Record phrase) SQL keyword
WHILE	WHILE	Option (DO, FOR EACH, REPEAT statements)
WHITE	W	Option (Color phrase)
WIDTH	WIDTH	Option (Frame phrase)
WITH	WITH	Keyword (Frame phrase) SQL keyword
WORK	WORK	SQL keyword
WORKFILE	WORKFILE	Keyword (DEFINE WORKFILE statement)
XCODE	XCODE	Option (COMPILE statement)
XREF	XREF	Option (COMPILE statement)
YEAR	YEAR	Function
YELLOW	Y	Option (Color phrase)
YES	YES	Logical value

Index