VERSION 6

DOCUMENTATION BULLETIN II

Dear PROGRESS User:

This bulletin describes Version 6 documentation changes. These changes may be due to product enhancements, documentation errors, or production errors. This booklet is meant to supplement your documentation set. Changes to existing pages are highlighted by black change bars (vertical lines) that run along the side of the pages in this booklet. We recommend that you pencil in the appropriate changes, if possible. The new pages do not contain change bars; they describe the new features.

This PROGRESS *Documentation Bulletin II* contains:

- New pages about the Greek Character Set feature of Version 6.2D.
- New pages about the Raw Datatype feature of Version 6.2D.
- New pages about the R-code Librarian feature of Version 6.2F.
- Change pages for the *PROGRESS 3GL Interfaces* manual — 1-26, 1-28, 3-1 through 3-3, 3-9.
- A change page for the *Test Drive* manual — 2-29.
- Change pages for the *System Administration I: Environments Guide* — 2-9 , 2-10, 2-11.
- Change pages for the *System Administration II: General* manual — 4-16, 4-17, 4-18, 4-23, 6-18.
- Change pages for the *PROGRESS Language Reference* — 109, 168, 314, 315.
- Chapter 16 of the *PROGRESS Language Tutorial.*

New Pages About the Greek Character Set Feature

(Version 6.2D)

2.1.1 User-Defined Language Rules

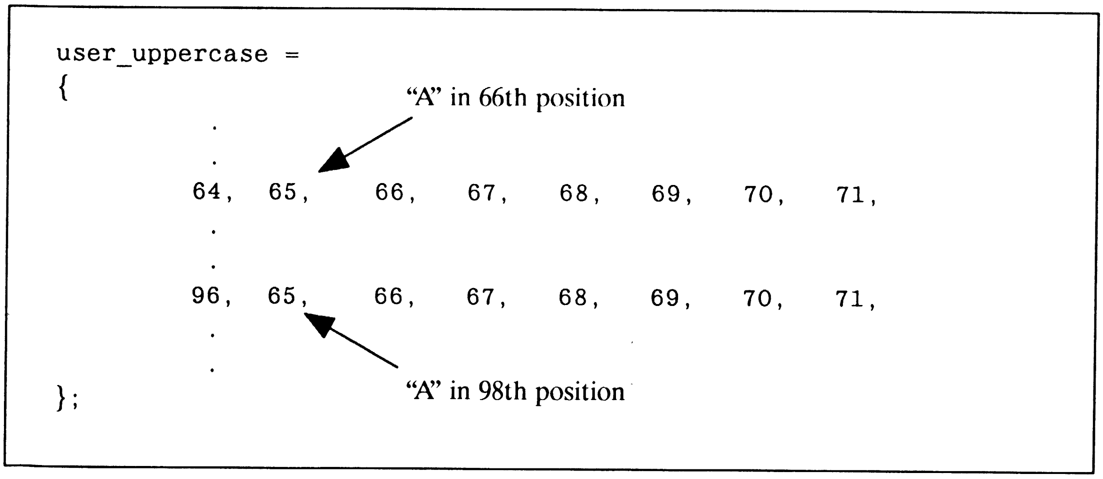
PROGRESS enables you to define your own language and control its collation sequence and case conversion. PROGRESS provides this capability to support languages outside the Basic, Danish, and Swedish groups. The Greek language, for instance, has unique collation requirements that the three language groups do not address.

Each "language" that PROGRESS supports is made up of the following four tables. To create your own language, you must prepare an ASCII text file that includes all four tables and then compile the file. The four tables, and the steps required to prepare them, are described in the following table.

Table 2–13: Language Tables

Table Name	Function
user_uppercase	Assigns an upper-case value to each ASCII character
user_lowercase	Assigns a lower-case value to each ASCII character
user_weight	Assigns a sort weight to each ASCII character
user_cs_weight	Assigns a case-sensitive sort weight to each ASCII character

user_uppercase Table. The following figure shows the format of a sample user_uppercase table:



```
user_uppercase =
{
                .
                .
        64,  65,   66,   67,   68,   69,   70,   71,
                .
                .
        96,  65,   66,   67,   68,   69,   70,   71,
                .
                .
};
```

"A" in 66th position
"A" in 98th position

There are 256 positions, ranging from 0 to 255, in all four tables. Each position stores an ASCII value. PROGRESS determines the upper-case value of a character by indexing the character against this table. For example, the character "a" has an ASCII value of 97. PROGRESS converts this character to the upper-case character "A," which has an ASCII value of 65, by going to the 98th position in the user_uppercase table. (Since the table starts at position 0, ASCII value 97 is indexed by the 98th position.)

Likewise, positions 99 through 104 hold the ASCII values 66 to 71 ("B" to "G").

user_lowercase Table. This table's format is identical to the `user_uppercase` table. However, instead of converting characters the their upper-case value, this table converts characters to their lower-case value.

To convert the upper-case character "A" (ASCII 65) to the lower-case character "a" (ASCII 97), PROGRESS goes to the 66th position of the table. The number 97 is in this position, and therefore PROGRESS converts "A" to "a."

user_weight Table. This table determines the order in which PROGRESS collates or sorts the characters. PROGRESS indexes the table in the same way as the other tables, using the ASCII value of each character. PROGRESS locates the sort weight of "a" (ASCII 97) by going to the 98th position in the table.

Generally, you will want upper-case and lower-case characters to sort together. Therefore, you will want to give "A" and "a" the same sort weight. This sort weight should be the ASCII value of upper-case "A," or 65. When PROGRESS indexes the `user_weight` table, it finds the number 65 in the 66th and 98th positions.

user_cs_weight Table. PROGRESS uses the `user_cs_weight` table to do case-sensitive sorting. The `user_cs_weight` table has the same format as all three of the other tables, but it enables you to give each character a unique sort weight. In this table, you want to give "A" and "a" different sort weights. You can make PROGRESS sort "A" using its ASCII value (65), and sort "a" using its ASCII value (97). In this example, "A" sorts before "a."

Building Your Tables. You can manipulate the tables to sort the characters in any order you want. However, when you build your tables, you must conform to the following rules:

- The tables must be in this order: `user_uppercase`, `user_lowercase`, `user_weight`, and `user_cs_weight`.
- Each table must have 256 positions separated by commas.
- Each cell in the table must contain an ASCII value.
- You must use the proper syntax

NOTE: The file procoll.eng in your installation directory is a sample version of a language table file.

You can use any ASCII text editor to prepare your tables. To make the tables more readable, you can place comments in the tables and represent the ASCII values four different ways:

- By typing the ASCII value in decimal.
- By typing the ASCII value in Hexidecimal (place a 0x before the hexidecimal number).
- By typing the actual ASCII character, surrounded by single quotes.
- By giving the ASCII value a name, using the #define directive.

Therefore, 88, 'X', and 0x58 all represent the same character.

Naming an ASCII Value. The first line in the following figure demonstrates how to assign a name to an ASCII value. (The /* and */ characters enclose a comment.)

```
#define AtildeU 199   /* A (Uppercase) with tilde "~" */
user_uppercase =
{
           .
           .
        192,  193,  194,  195,  196,  197,  AtildeU, AtildeU,
           .
           .
 };
```

Each name must begin with a letter. Also, each name must be unique to the first 16 characters.

In addition, there is one symbol that must have a predefined value, ProcessSSharp. You must define this symbol as either 1 or zero. Set the value to 1 (#define ProcessSSharp 1) to have PROGRESS convert the German SSharp character to a double "S." Set the value to 0 (#define ProcessSSharp 0) if your language tables do not support the German SSharp.

In the directory where you installed PROGRESS (usually DLC), there is a procoll.eng file. Read this file to see a completed set of language tables. This file also illustrates how you can use define statements and comments.

Activating Your Tables. Once build a file that contains your completed tables, you must compile it with the following command:

OS	Command to Create Binary File procoll.dat
UNIX DOS and OS2	`proutil -C collation-compiler` *language-source-file*
VMS	`PROGRESS/UT=COLLATION/INPUT_FILE=`*language-source-file*

where *language_source_file* is the name of the ASCII file containing your completed language tables. PROGRESS creates a binary file named `procoll.dat.`

Once you have a `procoll.dat` file, you can begin to use your language. Generally, PROGRESS reads `$DLC/procoll.dat` at startup. However, if you want to store `procoll.dat` in a different directory and you want to give it a different name, you can use the PROCOLL environment variable to direct PROGRESS to the file's new location. For example:

```
PROCOLL=/usr/foo/mylang.dat.
```

Enter the following command to specify your new language. You must specify your new language on an **empty** database.

OS	Command to Specify Database Language
UNIX DOS and OS2	`proutil` *database-name* `-C language user-defined`
VMS	`PROGRESS/UTILITIES=LANGUAGE_USER` *database-name*

Also, after you create an index and add records to your empty database, be careful about changing your user_weight table. Changing it could corrupt your indexes. If this happens, run the Index Rebuild utility to repair the corrupted indexes.

Multiple Databases. When you use multiple databases, each database can have its own language. If you compare records from two databases, PROGRESS uses the language of the first connected database. You can override this default with the -xc *language* startup option.

New Pages About the Raw Datatype Feature

(Version 6.2D)

13.1.1 Using the Raw Datatype

The PROGRESS raw datatype allows you to retrieve and manipulate raw data from non-PROGRESS databases. You can only use the raw data type in memory, you cannot use it to define fields in the schema. PROGRESS does not perform any conversion on the data, you receive the bytes as supplied by the non-PROGRESS database.

You can use the raw data type for numerous reasons — anytime you need to manipulate data from non-PROGRESS databases without having PROGRESS give that data the characteristics of a certain data type. For example, you can use the raw datatype to bring over data that has no parallel PROGRESS datatype. By using the raw data type statements and functions, PROGRESS allows you to bring data from any field into your procedure, manipulate it, and write it back to the non-PROGRESS database. The functions and statements give you the means to define raw datatype variables, write data into a raw variable, find the integer value of a byte, change the length of a raw variable, and perform logical operations. The following procedure demonstrates how to retrieve raw values from the database, how to put bytes into variables, and how to write raw values back to the database.

rawdemo1.p

```
/*You must run this procedure against a non-PROGRESS
 demo database.*/


DEFINE VAR r1 AS RAW.
DEFINE VAR i AS INT.

FIND FIRST cust.
r1 = RAW(name).
PUTBYTE(r1,1) = 115.
RAW(name) = r1.
DISPLAY name.
END.
```

This procedure first creates the variable r1 and defines it as a raw data type. Next, it finds the first customer in the database and with the RAW function, takes the raw value of the field name and writes it into the variable r1. The PUTBYTE statement then puts the ASCII value of "s" (115) into the first byte of r1. The RAW statement takes the raw value of r1 and writes it back to the database. Finally, the procedure displays the customer name. `Second Skin Scuba` becomes `second Skin Scuba`. The next procedure shows how you can pull bytes from a field.

rawdemo2.p

```
/*You must run this procedure against a non-PROGRESS
 demo database.*/


DEFINE VAR i AS INT.
DEFINE VAR a AS INT.

FIND cust WHERE cust-num = 27.
i = 1.
REPEAT:
      a = GETBYTE(RAW(name),i).
      DISPLAY a.
      IF a = -1 THEN LEAVE.
      i = i + 1.
END.
```

This procedure finds the customer with the customer number 27, and then finds the ASCII value of each letter in the customer name. To do this, it retrieves the bytes from the name one by one and places them into the variable a. The GETBYTE statement returns a -1 if the byte number you try to retrieve is greater than the length of the expression you are retrieving it from. The next procedure demonstrates how you find the length of a raw value and how to change length of a raw expression.

rawdemo3.p

```
/*You must run this procedure against a non-PROGRESS
  demo database.*/

DEFINE VAR r3 AS RAW.

FIND FIRST cust.
r3 = RAW(name).
DISPLAY LENGTH(r3) name WITH DOWN. /*length before change*/
DOWN.

LENGTH(r3) = 2.
DISPLAY LENGTH(r3) name./*length after change*/
```

This procedure simply finds the number of bytes in the name of the first customer in the database then truncates the number of bytes to two. The procedure first displays an `18` because the customer name `Second Skin Scuba` contains 18 bytes (the number of letters plus the null terminator). It then displays "`Se`" and a 3, because you truncated the name to two bytes and added a null terminator.

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

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*.

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 r1 AS RAW.

FIND customer WHERE cust-num = 26.
DISPLAY name.
r1 = RAW(name).
PUTBYTE(r1,1) = ASC('B').
PUTBYTE(r1,2) = ASC('i').
PUTBYTE(r1,3) = ASC('l').
PUTBYTE(r1,4) = ASC('l').

RAW(name) = r1
DISPLAY name.
```

This procedure finds the name of customer 26, Jack's Jacks, and stores it in the raw variable r1. The PUTBYTE statement replaces the first four bytes in the name with the specified ASCII values. The procedure then writes the values in r1 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

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

New Pages About the R–Code Librarian Feature

(Version 6.2F)

4.1 BUILDING LIBRARIES WITH THE PROLIB UTILITY

A library is a specially organized file that you can use to store all of your compiled procedures (`.r`, or object, files). Placing your object files in a library improves the performance of your applications for the following reasons:

Faster access to your files — Since all of your object files are stored in one place, the system finds them more easily.

Less open and close operations — The system only needs to open a library once, instead of opening each object file individually.

Smaller sort files — The system can easily read object files into memory from a library. Therefore, the system does not swap library object code into temporary sort files, like it does for other object files. Since many users can share the same library, the size of the temporary sort files is reduced for each user.

To use a library, you must first create the library using the `prolib` utility's `-create` option. The library name must have a `.pl` extension. Once you create the library, you can add files to the library using the `prolib` utility's `-add` option. To make the library accessible to PROGRESS, you must include the library on your PROPATH.

PROGRESS opens a library when it searches the PROPATH for a file and finds the file in a library. When opening a library, PROGRESS loads the library's internal directory into memory and keeps it in memory until all users accessing the library end their PROGRESS sessions. If you are accessing more than one library, PROGRESS loads each library's internal directory into memory.

On a DOS machine, the extra memory used by these internal directories may slow your performance. You can limit the size of the directories by using the Prolib–Memory (`-plm`) startup option.

If you change the PROPATH environment variable during a PROGRESS session, PROGRESS closes all of the libraries that are not included on the new PROPATH.

To create and use libraries, use the `prolib` utility:

Operating System	Prolib Command
UNIX DOS & OS/2	`prolib` *library-name* `[options]`
VMS	`PROLIB /[options]=`*library-name*

BTOS/CTOS	`PROGRESS Librarian`
	`Library Name` — `library-file-name`
	`[Create New Library -cr ]` — `yes/no`
	`[Add File List -add ]` — *file-name* [*file-name...*]
	`[Replace File List -rep ]` — *file-name* [*file-name...*]
	`[Delete File List -del ]` — *file-name* [*file-name...*]
	`[Listing of library -list ]` — *file-name* [*file-name...*]
	`[Extract File List -ext ]` — *file-name* [*file-name...*]
	`[Yank File List -yank ]` — *file-name* [*file-name...*]
	`[Compress -comp ]` — `yes/no`
	`[Supress warning -nowarn ]` — `yes/no`
	`[Verbose -verbose ]` — `yes/no`
	`[Options]`

You can use the following major options with the `prolib` utility:

`-create`	Creates a new library.
`-add`	Adds a file or files to a library.
`-replace`	Replaces a file or files in a library.
`-delete`	Deletes a file or files from a library.
`-list`	Lists the table of contents for a library.

`-extract`	Extracts a file or files from a library. This option copies a file from the library into another file, outside of the library, and gives it the same name. If you add a file to a library by specifying its pathname, the `prolib -extract` option extracts the file and places it in the directory specifed in the pathname.
`-yank`	Works like the `-extract` option, but places all copied files into your current working directory.
`-compress`	Compresses the library by making a copy of it. For large libraries, there may not be enough disk space for this option.

In addition to these major options, you can use the following options to further control the way each major option is processed:

`-nowarn`	Suppresses any warning messages that may occur during the operation of the major options. If you add a file with the `-add` and `-nowarn` options, and the file already exists in the library, `prolib` replaces the file. Conversely, if you replace a file with the `-replace` and `-nowarn` options, and the file does not exist, `prolib` adds the file.
`-pf`	Enables you to supply command-line arguments in a parameter file. When adding command-line arguments to a file, you must use UNIX-style syntax. Note that you cannot include the `-pf` option in a parameter file.
`-verbose`	Directs `prolib` to display processing information that is ordinarily suppressed.
`-date`	Specifies the format of the date as it appears when you use the `-list` option.

When specifying an option, you do not have to type the complete option name. You can type the minimally unique string for each option (for example, `-l` for `-list` and `-e` for `-extract`).

NOTE: If you are using VMS as your operating system, you can specify only one major option at a time. However, you can combine major options in a parameter file, although you must specify the options using UNIX-style syntax. Also, you can use wildcards for all of the `prolib` options, but you cannot use wildcards in a parameter file with the `-add` and `-replace` options.

4.1.1 Creating a Library

To create a library, use the following command:

Operating System	To Create a Library	
UNIX DOS & OS/2	`prolib` *library-name* `-create`	[`-nowarn` `-verbose` `-pf` *file-name*]
VMS	`PROLIB/CREATE=`*library-name*	[`/NOWARN` `/PARMFILE=`*file-name* `/VERBOSE`]

BTOS/CTOS	`PROGRESS Librarian`
	`Library Name`
	`[Create New Library -cr ]`
	⋮
	`[Options]`
		
		`library-file-name`
		`yes/no`

where *library-name* is the name of the library you want to create.

All libraries must have a .pl extension. You can add libraries to your PROPATH environment variable by using fully qualified pathnames (absolute pathnames), partially qualified pathnames (relative pathnames), or using the library name by itself. If you use the library name by itself, PROGRESS assumes that the library is located in your current working directory.

4.1.2 Adding a File to a Library

To add a file or files to a library, use the following command:

Operating System	To Add a File to a Library	
UNIX DOS & OS/2	`prolib` *library-name* `-add` *file-name* [*file-name...*]	[`-nowarn` `-verbose` `-pf` *file-name*]
VMS	`PROLIB/ADD=`*library-name* *file-name* [*file-name...*]	[`/NOWARN` `/PARMFILE=`*file-name* `/VERBOSE`]

BTOS/CTOS	`PROGRESS Librarian`
	`Library Name`
	⋮
	`[Add File List -add ]`
	⋮
	`[Options]`
		
		`library-file-name`
		
		file-name [*file-name...*]

where *library-name* is the name of the library you want to add the file to, and *file-name* is the name of the file. You can add a file to a library by specifying the file's absolute pathname, relative pathname, or, if the file is in your current working directory, by specifying the filename itself.

When you add a file to a library by specifying an absolute or relative pathname, the library entry for that file retains the entire pathname. For example, if you enter a file into a library called newlib.pl using the following command:

```
prolib newlib.pl -add /usr/apps/proc1.r
```

the pathname /usr/apps/proc1.r appears in the library's table of contents. You can display the library's table of contents with the -list option. When you extract a file from a library, prolib copies the file into the directory specified by the pathname (in this example /usr/apps). See "Extracting a File in a Library" in this section for more information.

NOTE: You can use a library to store the Data Dictionary's object files, which are stored in the prodict subdirectory of your installation directory (/DLC by default). Storing these object files in a library should improve the Data Dicitonary's performance by approximately 10%. To store the files in a library, perform the following steps:

- Change to the directory where you installed PROGRESS.
- Create a library called prodict.pl.
- Add all of the object files (.r) that are in the prodict subdirectory of your installation directory into the library prodict.pl.

  After you have added the object files into the library prodict.pl, you can use the -list option to see if all of the files have been added correctly. If they have, you may want to delete them from the prodict subdirectory to save disk space.

If you try to add a file that already exists, prolib displays an error message. However, if you specify the -nowarn option, prolib replaces the existing file. See "Replacing a File in a Library" in this section for more details.

4.1.3 Replacing a File in a Library

To replace a file or files in a library, use the following command:

Operating System	To Replace a File to a Library	
UNIX DOS & OS/2	`prolib` *library-name* `-replace` *file-name* [*file-name...*]	[`-nowarn` `-verbose` `-pf` *file-name*]
VMS	`PROLIB/REPLACE=`*library-name* *file-name* [*file-name...*]	[`/NOWARN` `/PARMFILE=`*file-name* `/VERBOSE`]

BTOS/CTOS	`PROGRESS Librarian`
	`Library Name`
	`[Replace File List -rep ]`
	`[Options]`
		`library-file-name`
		
		file-name [*file-name...*]

where *library-name* is the name of the library with the file you want to replace, and *file-name* is the name of the file. When you replace a file, the system overwrites the old file a file of the same name. You can replace a file in a library by specifying the file's absolute pathname, relative pathname, or, if the file is in your current working directory, by specifying the filename itself. If you try to replace a file that does not exist, `prolib` displays an error message. However, if you specify the `-nowarn` option, `prolib` adds the existing file to the library. See "Adding a File to a Library" in this section for more information.

4.1.4 Deleting a File from a Library

To delete a file or files from a library, use the following command:

Operating System	To Delete a File from a Library	
UNIX DOS & OS/2	`prolib` *library-name* `-delete` *file-name* [*file-name*]	[`-nowarn` `-verbose` `-pf` *file-name*]
VMS	`PROLIB/DELETE=`*library-name* *file-name* [*file-name...*]	[`/NOWARN` `/PARMFILE=`*file-name* `/VERBOSE`]

BTOS/CTOS	`PROGRESS Librarian`
	`Library Name`
	`[Delete File List -del ]`
	`[Options]`
		`library-file-name`
		
		file-name [*file-name...*]

where *library-name* is the name of the library with the file you want to delete, and *file-name* is the name of the file. You can specify more than one file at a time.

4.1.5 Listing the Contents of a Library

To list the contents of a library, use the following command:

Operating System	To List the Contents of a Library
UNIX DOS & OS/2	`prolib` *library-name* `-list` *file-name* [*file-name...*] [`-nowarn` | `-verbose` | `-pf` *file-nam* | `-date` *date-format*]
VMS	`PROLIB/LIST=`*library-name* *file-name* [*file-name...*] [`/NOWARN` | `/PARMFILE=`*file-name* | `/VERBOSE` | `/DATE` *date-format*]

BTOS/CTOS	`PROGRESS Librarian`
	`Library Name` `library-file-name`
	⋮
	`[Listing of library -list ]` *file-name* [*file-name...*]
	⋮
	`[Options]`

where *library-name* is the name of the library with the contents you want to list. If you specify a *file-name,* `prolib` lists information about that file only. You can specify more than one file at a time. In addition, you can use the ? and * wildcard symbols to specify files (on the UNIX command line, you need to place the escape character (\ ) before a wildcard character).

The output of the -list option is:

`Name` — The filename of each file in the library.

`Size` — The size, in bytes, of each file in the library. Each `.r` file increases slightly in size when you add them to a library, because `prolib` appends a tag to the end of the file.

`Type` — The file type of each file in the library. Prolib recognizes two file types: `R` (`.r` files), and `O` (any other type file). Although libraries are specifically designed to hold `.r` files, you can place any type of file in them.

`Offset` — The distance, in bytes, of the start of each file from the beginning of the library.

`Modified` — The date and time of each file's last modification.

`Added To Lib` — The date and time of each file's entry into the library.

You can also change the format of the dates as they appear in the `Modified` and `Added To Lib` fields with the `-date` option, where *date-format* is the three letters `m` (month) `d` (day) and `y` (year). To specify the format you want, place the letters in the order you want the months, days, and years to appear on screen. The default is `mdy`.

4.1.6 Extracting Files from a Library

To extract a file or files from a library, use the following command:

Operating System	To Extract a File from a Library
UNIX DOS & OS/2	`prolib` *library-name* `-extract` *file-name* [*file-name...*] [`-nowarn` | `-verbose` | `-pf` *file-name*]
VMS	`PROLIB/EXTRACT=`*library-name* *file-name* [*file-name...*] [`/NOWARN` | `/PARMFILE=`*file-name* | `/VERBOSE`]

BTOS/CTOS	`PROGRESS Librarian`
	`Library Name` library-file-name
	⋮
	`[Extract File List -ext ]` *file-name* [*file-name...*]
	⋮
	`[Options]`

where *library-name* is the name of the library with the file you want to extract, and *file-name* is the name of the file. You can specify more than one file at a time.

When you extract a file from a library, `prolib` copies the file and places it in its original directory. Depending on how you add a file to a library, `prolib` extracts differently:

- If you added the file by specifying its absolute pathname, `prolib` searches for the original directory. If the directory does not exist, `prolib` displays an error message. You can specify the -yank option instead of `-extract` to strip everything but the filename from the pathname and place the file in your current directory (see "Using the -Yank Option" in this section for more information).

- If you added the file by specifying a relative pathname, `prolib` searches your current working directory for the subdirectory indicated in the pathname. For example, if you added a file by using the relative pathname `apps/proc1.r`, `prolib` searches your current working directory for the subdirectory `apps`.

  If the directory does not exist, prolib displays an error message. You can specify the `-yank` option instead of `-extract` to strip everything but the filename from the pathname and place the file in your current directory.

- If you added the file by specifying the file's filename, `prolib` places the file into your current working directory.

If the file you are extracting (or yanking) already exists in the directory outside of the library, `prolib` displays the following message:

```
File file-name already exists.  Do you want to replace it?
Please answer 'yes' or 'no' or 'rename to assign new name:
```

where *file-name* is the pathname of the file you are extracting (or yanking). If you type yes, prolib overwrites the file. If you type no, prolib does not overwrite the file. If you type rename, prolib displays the following message:

```
Enter new name to use for extracted file file-name-->
```

where *file-name* is the pathname of the file you are extracting (or yanking). After you enter the pathname, prolib gives the file the new pathname and places it in the appropriate directory.

4.1.7 Using the -yank Option

To extract a file or files from a library using the -yank option, use the following command:

Operating System	To Extract a File from a Library
UNIX DOS & OS/2	prolib *library-name* -yank *file-name* [*file-name...*] [-nowarn | -verbose | -pf *file-name*]
VMS	PROLIB/YANK=*library-name* *file-name* [*file-name...*] [/NOWARN | /PARMFILE=*file-name* | /VERBOSE]
BTOS/CTOS	PROGRESS Librarian Library Name : library-file-name [Yank File List -yank] : *file-name* [*file-name...*] [Options]

where *library-name* is the name of the library with the file you want to extract, and *file-name* is the name of the file. You can specify more than one file at a time.

When extracting a file with the -yank option, prolib discards everthing but the filename from the file's pathname. For example, if you added a file using the pathname /usr/apps/proc1.r, prolib discards /usr/apps from the pathname and places proc1.r in your current working directory.

4.1.8 Compressing a Library

To compress a library, use the following command:

Operating System	To Compress a Library	
UNIX DOS & OS/2	`prolib` *library-name* `-compress`	[`-nowarn` `-verbose` `-pf` *file-name*]
VMS	`PROLIB/COMPRESS=`*library-name*	[`/NOWARN` `/PARMFILE=`*file-name* `/VERBOSE`]

BTOS/CTOS	`PROGRESS Librarian`
	`Library Name`
	⋮
	`[Compress -comp ]`
	⋮
	`[Options]`
		`library-file-name`
		
		`yes/no`

where *library-name* is the name of the library you are compressing. When you use this command, `prolib` makes a copy of the library. The `-compress` option removes extra spaces that occur in the library as a result of repeated adds or deletes.

To make a copy of the library, `prolib` creates a temporary file that requires an area of disk space equal to the size of the library. Before compressing a library, make sure you have enough disk space for the temporary file. Temporary files have names that begin with the letters PLB, followed by numbers and possibly letters.

If your system goes down during a compress operation, the temporary file may remain on your disk. Delete this file to free up your disk space.

3.1.1 Prolib-Memory

SYNTAX		
	DOS OS/2 UNIX	`-plm`
	VMS	`/LIBMEM`
	BTOS CTOS	`[options] -plm`

▨	Use With	Max Value	Min Value	Single-User Default	Multi-User Default
▨	P,M	▨	▨	▨	▨

When you use PROGRESS libraries (see "Building Libraries with the Prolib Utility" in Chapter 4 for more information), the library's internal directory is loaded into memory. The system uses this directory to access the object files stored in the library. If you have a large library, you may want to use the Prolib-Memory (`-plm`) option to allocate a 512 byte cache for the library's directory, instead of having the entire directory loaded into memory. This option slows the library's speed of access but increases available memory.

3.1.2 Prolib-Swap

SYNTAX		
	DOS OS/2 UNIX	`-pls`
	VMS	`/LIBSWAP`
	BTOS CTOS	`[options] -pls`

▨	Use With	Max Value	Min Value	Single-User Default	Multi-User Default
▨	P,M	▨	▨	▨	▨

When accessing object files stored in a library (see "Building Libraries with the Prolib Utility" in Chapter 4 for more information), PROGRESS reads the files directly from the library, instead of swapping the files into temporary sort files. However, if you are running PROGRESS over a network and you place your library on a file server, remote reads from the library may take longer than reading an object file once and storing it locally in a temporary sort file. The Prolib-Swap (-pls) option enables you to override the default and have PROGRESS store the object files in temporary sort files.

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

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

PROGRESS 3GL Interface Guide Documentation Changes

(Version 6.2)

PMLOAD.BAT invokes four commands to prepare your PROGRESS executable:

- LINK (from Microsoft) to link your object modules and libraries.
- EXEMOD (from Microsoft) to adjust the size of the runtime stack (in bytes, hexadecimal notation).
- EXPRESS (from Ergo OS/286 kit) to produce .EXP (program) and .XMP (map) files from a .EXE.
- BIND (from Ergo) to bind the OS/286 kernel together with PROGRESS and create a new executable .EXE file. Note that BIND is distributed separately from the standard Eclipse developer's kit. Therefore, this command may have no effect when you run PMLOAD.BAT. If you do not have the BIND command, PMLOAD.BAT creates a .EXP—not a final executable .EXE—file. In this case, to run PROGRESS, you invoke the .EXP using the OS/286 UP command at the DOS prompt (UP *executable*.EXP), or you can run it directly from the OS/286 command processor.

For more information regarding any of these commands, see your Microsoft or Eclipse OS/286 documentation.

NOTE: If you are using HLC, HLI, or a non-PROGRESS gateway, then **prior to running PMLOAD.BAT**, you must run the OS/286 PATCH command to patch the Microsoft composite C library LLIBCE.LIB. Furthermore, you must rename (or copy) LLIBCE.LIB to AIALIBCE.LIB, and finally, the environment variable AIALIB must point to AIALIBCE.LIB. Before using the PATCH command, you should add the OS/286 installation directory to your PATH, and set the OS286 environment variable to point to this installation directory (C:\OS286, by default). The following .BAT illustrates the necessary steps; read it carefully.

PMSETUP.BAT

```
REM Add OS/286 install dir to PATH and set OS286 var.
PATH=%PATH%;\OS286
SET OS286=C:OS286

REM Move to Microsoft Library install dir.
CD C:/MSC/LIB

REM Run PATCH on LLIBCE, rename it, and set AIALIB var.
PAUSE Insert OS/286 system diskette in drive A:
      Press any key when ready.
A:PATCH MSC5E
COPY LLIBCE.LIB AIALIBCE.LIB
SET AIALIB=C:/MSC/AIALIBCE.LIB
SET PROOVL=E:\OVLDIR
```

DOS Protected Mode: There are two cases:

(1) You have the OS/286 BIND command. PMLOAD.BAT generates an executable .EXE file, which you can execute directly.

(2) You do not have OS/286 BIND. In PMLOAD.BAT, the OS/286 EXPRESS command converts your executable (.EXE) to .EXP (program) and .XMP (map) files. Now, to run PROGRESS, invoke the .EXP using the OS/286 UP command at the DOS prompt (UP *executable*.EXP), or run it directly from the OS/286 command processor.

If your attempt to run protected mode PROGRESS fails with kernel configuration errors, run the TUNE.EXE utility (supplied with PROGRESS), to automatically reconfigure your kernel.

NOTE: If you use a 286 machine, TUNE may crash your machine as it proceeds through a series of system tests. This is harmless; simply restart TUNE.

See also Chapter 2 in *System Administration I: Environments.*

1.2.2 Running BTOS/CTOS Executables

Non–HLI Executables: You have three alternatives:

- Rename the .RUN file produced by the link submit file to PROGRESS.RUN in the directory where you installed PROGRESS ([sys]<DLC>, by default).
- Use the command file editor to change the run file for the PROGRESS 4GL command to be the newly created .RUN file.

or

- Use the command file editor to create a new command by copying the PROGRESS 4GL command and supplying the new run file for the new command.

HLI Executables: Use the CTOS RUN command to execute the new .RUN file, or create a new command using Command File Editor. If you want to pass parameters, you must handle these in your mnin() function.

Chapter 3 General HLC Programming Considerations

This chapter deals with general programming considerations for using HLC, as well as specific issues of data type conversion, the PROGRESS CALL statement, and the HLC dispatch routine. Included are examples for using the CALL statement and modifying the dispatch routine.

3.1 USING THE CALL STATEMENT

You use the PROGRESS 4GL CALL statement to call a C program from a PROGRESS procedure. For a description of how the CALL statement works, see section 2.3.

The CALL statement has the following syntax:

SYNTAX

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

routine-name

The name the dispatch routine (PRODSP) maps to an actual C function. Note that *routine-name* must have the same letter case as its declaration in the dispatch routine. PROGRESS examples all use upper case.

argument

One or more arguments that you want to pass to the C function. If you supply multiple arguments, separate them with spaces. (If you use other delimiters—commas for example—they are passed as arguments.)

Note that PROGRESS converts all arguments to character strings before passing them to a C routine (they are passed as an array of character pointers). Therefore, your C routines should expect null-terminated character strings and do data conversions as necessary.

If *routine-name* is not listed in the PRODSP dispatch routine (or is listed in a different letter case), an error results.

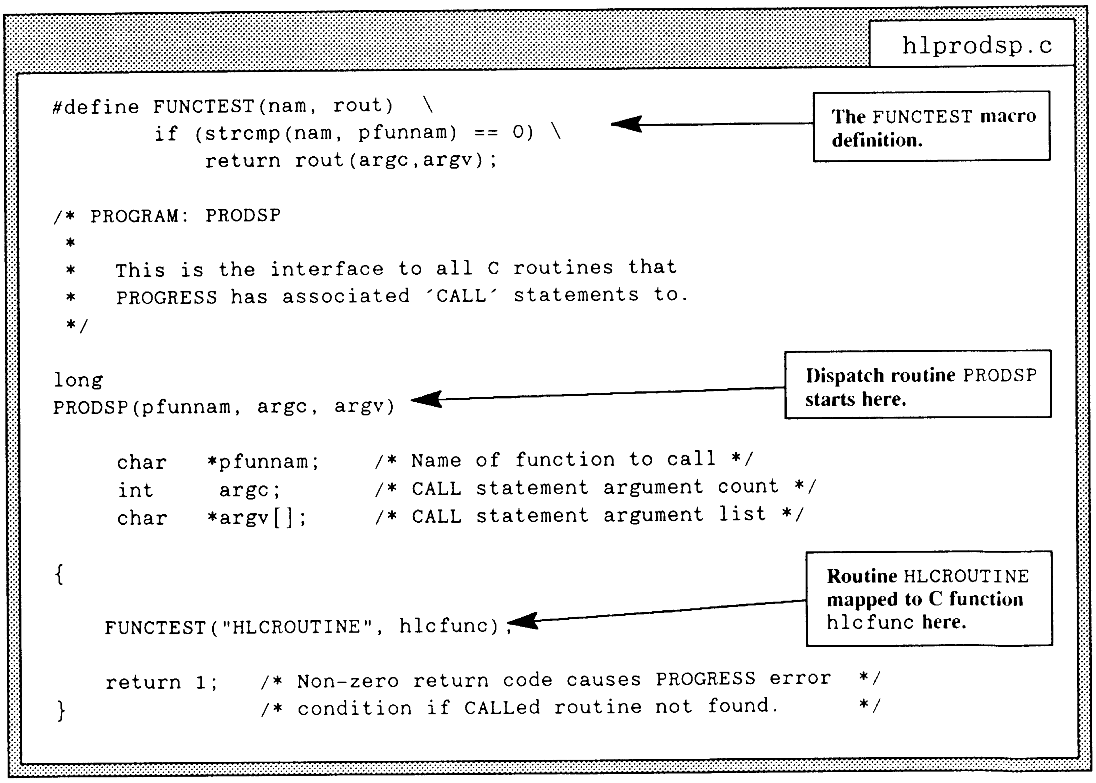
When you use a CALL statement to invoke a routine that updates a shared buffer, you must make sure that a transaction is active at the time of the call. See Section 3.6.6 for more information.

3.2 THE HLPRODSP.C DISPATCH ROUTINE FILE

The `hlprodsp.c` dispatch routine file contains the PRODSP dispatch routine. The PRODSP dispatch routine is where you define routine identifiers and their corresponding C routine names.

NOTE: The `/dlcload/hlc` directory contains a prototype `hlprodsp.c` dispatch routine file. Do not modify this file; instead, copy it to a working directory and modify the copy.

The following listing of `hlprodsp.c` shows routine identifier HLCROUTINE being mapped to C function `hlcfunc`:


hlprodsp.c
The FUNCTEST macro definition.
Dispatch routine PRODSP starts here.
Routine HLCROUTINE mapped to C function hlcfunc here.

```
#define FUNCTEST(nam, rout)  \
        if (strcmp(nam, pfunnam) == 0) \
            return rout(argc,argv);

/* PROGRAM: PRODSP
 *
 *   This is the interface to all C routines that
 *   PROGRESS has associated 'CALL' statements to.
 */

long
PRODSP(pfunnam, argc, argv)

     char   *pfunnam;      /* Name of function to call */
     int     argc;         /* CALL statement argument count */
     char   *argv[];       /* CALL statement argument list */

{

    FUNCTEST("HLCROUTINE", hlcfunc);

    return 1;     /* Non-zero return code causes PROGRESS error  */
}                 /* condition if CALLed routine not found.      */
```

For each routine you want to add, include a FUNCTEST line. For example, if you want to map two routine names (HLCROUTINE1 and HLCROUTINE2) to two corresponding C functions (hlcfunc1 and hlcfunc2), include the following lines in PRODSP:

```
FUNCTEST("HLROUTINE1", hlfunc1);
FUNCTEST("HLROUTINE2", hlfunc2);
```

NOTE: HLROUTINE1 is in upper case, so use upper case for the CALL statement since the routine name for the CALL statement and FUNCTEST declaration must use the same letter case.

When you compile hlprodsp.c, the C compiler translates the FUNCTEST macro references to C language code. For example, the following line

```
FUNCTEST("HLROUTINE", hlfunc);
```

translates to:

```
if (strcmp("HLROUTINE", pfunnam) == 0)
    return hlfunc(argc,argv);
```

3.2.1 Syntax of FUNCTEST Macro

The FUNCTEST macro that appears in the prototype hlprodsp.c dispatch routine file has the following syntax:

SYNTAX

FUNCTEST("*routine-identifier*", *function-name*);

routine-identifier

A string enclosed in quotes. Letter case must match that used in the CALL statement.

function-name

The name of your C routine.

NOTE: Using the FUNCTEST macro forces the top-level C routines you call from the PRODSP dispatch routine to have the following form:

```
int function-name(argc,argv)
    int    argc;
    char   *argv[];
```

If you want your top-level C routines to have a different form, don't use the FUNCTEST macro, or rewrite it to suit your particular needs.

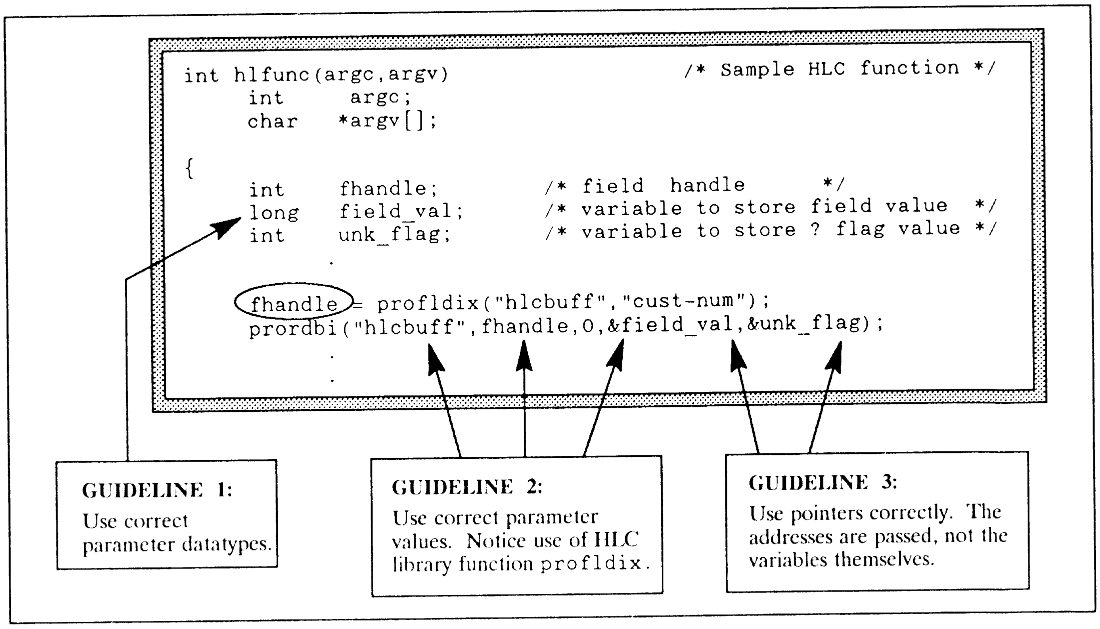
The prordbi function has the following declaration:

```
int prordbi (pbufnam, fhandle, index, pvar, punknown)
   char   *pbufnam;
   int    fhandle;
   int    index;
   long   *pvar;
   int    *punknown;
```

The parameters shown in the declaration have the following uses:

pbufnam	Points to the shared buffer name.
fhandle	This input parameter is the field handle returned by profldix.
index	If the field is an array field (i.e., if it has multiple extents), index specifies which element in the array to read. If the field is not an array field, set index to 0.
pvar	Points to the value of the integer field.
punknown	Points to a value of 1 if the integer field has the unknown (?) value, and a value of 0 otherwise.

Figure 3–2 shows a call being made to prordbi that illustrates HLC programming guidelines.



```
int hlfunc(argc,argv)                        /* Sample HLC function */
    int      argc;
    char     *argv[];

{
    int      fhandle;            /* field  handle        */
    long     field_val;          /* variable to store field value  */
    int      unk_flag;           /* variable to store ? flag value */
                .

    fhandle = profldix("hlcbuff","cust-num");
    prordbi("hlcbuff",fhandle,0,&field_val,&unk_flag);
                .
                .
```

GUIDELINE 1:
Use correct parameter datatypes.

GUIDELINE 2:
Use correct parameter values. Notice use of HLC library function profldix.

GUIDELINE 3:
Use pointers correctly. The addresses are passed, not the variables themselves.

Figure 3–1: An Example HLC Library Function Call

PROGRESS Test Drive Documentation Changes

(Version 6.2)

display an error message even when it can't find a record. Add NO-ERROR to the FIND statement:

```
    PROMPT-FOR sales-rep.
    FIND salesrep USING sales-rep NO-ERROR.
```

End the FIND statement with a period and press RETURN to move the cursor to the next line in the editing area.

Now, let's suppose that the record does not exist in the database. Remember the figure above that showed how you would want this to work in a manual system? You want to create a new record and assign the sales rep's initials to that record as well as supply all the other pertinent information. Type these lines into the edit area:

```
    PROMPT-FOR sales-rep.
    FIND salesrep USING sales-rep NO-ERROR.
    IF NOT AVAILABLE salesrep THEN DO:
       CREATE salesrep.
       ASSIGN sales-rep.
       UPDATE slsname slsrgn slstitle slsquota hire-date
              WITH 1 COLUMN 1 DOWN.
    END.
```

Note that many of the lines in this procedure are indented. The indentation is purely for convenience. Indentation and formatting have no effect at all on how the procedure runs.

Here, if a record for the sales rep does not exist, PROGRESS creates one and assigns to, or attaches to, that record the sales rep's initials that were supplied in the PROMPT-FOR step. Then the user is allowed to supply the rest of the sales rep information: name, region, title, quota, and date hired.

Now, what if the sales rep record *does* already exist? In that case, you probably want to allow the user to make changes to the information in the record. To do so, add the following lines:

System Administration I: Environments Documentation Changes

(Version 6.2)

Because DOS does not support multiple terminal types, PROGRESS terminal definition files on DOS serve only to support the OUTPUT TO *stream* MAP *termcapfile-entry* and INPUT FROM *stream* MAP *termcapfile-entry* 4GL statements. An entry in the terminal definition file can contain IN( ) and OUT( ) lines that map character values. Typically, character mapping is used to support non-English character sets. See also "Extended Alphabet Support" in Chapter 2 of the *Programming Handbook.*

2.1 SPECIAL DOS CONSIDERATIONS

The following sections contain information about special DOS features and files.

2.1.1 Protected Mode PROGRESS

In *protected mode,* a program executes in RAM above the one-megabyte address boundary. This allows you to overcome the memory limitations of the standard DOS operating system. Protected mode PROGRESS runs on most 80286- or 80386-based systems that have a minimum 2MB of RAM installed and that have a compatible BIOS. Known compatible BIOS implementations includes those by Phoenix, Award, and IBM.

NOTE: Protected mode PROGRESS does not run on the following systems: Televideo, and systems equipped with an Arc 286 Turbo card or an Orchid PC Turbo card.

When you install PROGRESS, two executables are automatically installed, _PROGRES (Memory Saver) and _PPROGRS (Protected Mode). To run an executable, you must set the PROEXE environment variable to the executable you want to run (by default, PROEXE is set to run the Memory Saver executable, _PROGRES). If you run Protected Mode PROGRESS, the _PPROGRS executable requires 1.15 MB of RAM, which includes a small portion of the 640 KB of memory addressed directly by DOS. You must configure the remaining memory as extended memory; it cannot be expanded memory. Check your hardware documentation to ensure that any dip switches on your memory card(s) have been set so that the system recognizes the on-board memory.

The amount of available extended memory is usually displayed when you boot your machine. The WHATMEM utility, supplied with PROGRESS, approximates how much extended memory you have, how much extended memory is currently in use, and how much extended memory is available for protected mode. WHATMEM is located in the \DLC directory and is entered at the DOS system prompt.

If your system only has 2MB of RAM, do not use this memory for any virtual disks. Protected mode PROGRESS requires that approximately 1.5MB of RAM is available at all times. Executing any programs that access extended memory could lead to unpredictable results. For example, do not execute protected mode PROGRESS or Lotus 1-2-3 (with its extended memory setting enabled) in a DOS shell.

Most 80286 and 80386-based machines can run Protected Mode PROGRESS without setting up a special system configuration file called `CONFIG.286`. However, PROGRESS supplies a `CONFIG.286` that is located in your installation directory. You can edit CONFIG.286 using any ASCII editor. Table 2-2 describes the configuration statements in `CONFIG.286`.

Table 2–2: Statements in CONFIG.286

Statement	Default	Comment
ENVSIZE=*n*	128	Set environment size to *n* bytes.
STARTMEG=*nn*	Lowest free memory.	Address in memory to load PROGRESS. Automatically set under most conditions. Use the WHATMEM utility.
PSPSWITCH=*n*	0	Determines the level of protected mode tasks (open files). Set this switch to 0 to override the protected mode limit of 20 open files. This increases the number of files you can open in PROGRESS, permits the DOS escape statement to work, and lets you split your database into multiple volumes.
LOWALLOCATE=*n*	128	Allocates *n* KB for Protected Mode program in low memory.

In most circumstances, your system can automatically determine the `STARTMEG` setting. You have to use the `STARTMEG` statement if there are other programs already accessing memory and your system cannot determine the address in memory to load PROGRESS.

Since PROGRESS needs at least 1.15 MB (one MB extended and 150 KB real) to run in protected mode, the `LOWALLOCATE` variable is set to 128. `LOWALLOCATE` assigns memory below 640 KB for use with protected mode programs.

`LOWALLOCATE` only allocates unused memory (free memory).

If you have 2 MB or more of extended memory, you do not need to use `LOWALLOCATE`.

Before PROGRESS Version 6, you could only have 20 system files open at once when running Protected Mode PROGRESS. Since DOS takes five of these for itself, your application is left with a maximum of fifteen.

However, the `PSPSWITCH` environment variable in your CONFIG.286 file enables you to open more than 20 files.

You must set the AIA environment variable to point to the location of CONFIG.286 that PROGRESS supplies. You can set AIA in your AUTOEXEC.BAT file as follows:

```
SET AIA=%DLC%\CONFIG.286
```

2.2 CUSTOMIZING DOS FOR PROGRESS

With the AUTOEXEC.BAT file, you can boot your computer, initialize your environment variables, start up your database, and backup the system without typing in a command. This is done using the DOS Batch language. A sample DOS Batch file is shown below:

```
REM /* Autoexec.bat */
SET DLC=C:\DLC
SET PATH=%DLC%;C:\UTILS;C:\BAT
SET PROPATH=;C:\APPL\PROC;C:\APPL\PROC1
_pprogrs dbname  -1 -e 55 -l 40 -D 50 -T d:\ -v 0f2f4f -p menu.p
ECHO "Your db will now be backed up"
CALL BACKUPDB.BAT
```

System Administration II: General Documentation Changes

(Version 6.2 and 6.2F)

To restore a database on BTOS/CTOS, use the command from the table above and enter the full name of the file that contains the backup. For example:

```
[F0]<SYS>backup
```

Important Rules for Restoring Incremental Backups. You must restore an incremental database backup to an existing database.

You must restore a database in the order in which you backed it up. You must first restore the full backup, followed by the first incremental backup, followed by the second incremental backup, and so on. If you try to restore a database out of sequence, you get an error message and the restore operation fails.

If you lose the second incremental and you used an overlapping factor of 1, then the third incremental will correctly restore the data lost in the second.

After you restore a full backup, do not update database records if you wish to restore successive incremental backups. If you make any database changes or even start a transaction, before completely restoring all backups, any successive, incremental backups (that were not restored) will be invalid unless you restart the restore procedure beginning with the full backup.

As you begin the restore procedure for a database, a report appears on your terminal that indicates:

- The sequence number of the volume currently being restored, and the total number of volumes from which the database is being restored.
- The name of the directory path and input device from which the data is being restored, and the name of the database that was backed up.
- The date of the backup.
- The number of blocks required to restore the database. This number includes both the data blocks that were backed up and the free blocks required to do the restore.

If a system failure occurs while you are restoring the database, start the restore operation again, beginning with the backup volume that you were restoring at the time of the system failure.

4.1.1 Using the prorest Utility: DOS Example

Company X sales administrator wants to restore his sales . db database from a full backup. He uses the following command.

```
prorest newsales.db a:\saleback
```

The database newsales.db is a new database that you create with the prorest command. The name of the device is `a:\saleback` (a floppy diskette drive) from which the full backup is being restored. The following figure shows the report John Doe sees on his terminal as the restore begins.

```
        Processing volume 1.
        a:full is a full backup of sales.db.
        This backup was taken Tue Aug 10 22:27:49 1987.
        It will require a minimum of 38 blocks to restore.
```

Figure 4–1: Report from a Full Restore of sales.db

The full backup was stored on three volumes (floppy diskettes). The report indicates the current volume being processed. After volume 1 is restored, John is prompted for the remaining volumes.

If John wants to restore an incremental backup of his database, he must first restore a full backup. To restore the first incremental backup a:\saleincl, he uses this command:

```
        prorest newsales.db a:\saleincl
```

The following figure shows the report he gets on his terminal as the restore operation begins.

```
    Processing volume 1.
    a:saleincl is a incremental backup of sales.db.
    This backup was taken Tue Aug 11 22:28:15 1987.
    It is based on the full backup of Tue Aug 11 22:27:49 1987.
    It will require a minimum of 38 blocks to restore.
```

Figure 4–2: Report from the First Incremental Restore of sales.db

If John restores a second incremental backup (we will assume it is called `a:\saleinc2`), he sees the report below.

```
Processing volume 1.
a:salesinc2 is a incremental backup of sales.db.
This backup was taken Tue Aug 12 22:30:13 1987.
It is sequence 2, based on the incremental of
                                   Tue Aug 11 22:28:15 1987.
It will require a minimum of 38 blocks to restore.
```

Figure 4–3: Report from the Second Incremental Restore of sales.db

After the restore is complete, newsales.db is ready to be used with PROGRESS.

4.1.2 The prorest Utility: UNIX

The database administrator of the Company X Development department wants to restore the doc.db database from a full backup. He uses the following command:

```
prorest newdoc.db /dev/rrm/0m
```

The database newdoc.db is a new database. The name of the device is /dev/rrm/0m (a 9-track tape drive) from which the full backup is being restored. The following figure shows the report the administrator sees as the restore begins.

```
Processing volume 1.
/dev/rrm/0m is a full backup of /usr/develop/doc.db.
This backup was taken Wed Aug 12 08:39:41 1987.
It will require a minimum of 64 blocks to restore.
```

Figure 4–4: Report from a Full Restore of doc.db

This command restores the database doc.db from a tape to newdoc.db. The report indicates that 1 volume is being processed.

4.1.1 Before-Image File Cluster Size

By default, the before-image file (`.bi`) cluster size is 16KB. Raising the cluster size with the `bi` option may improve system performance, but at the cost of increased disk space usage. The `bi` option has the following syntax:

Operating System	To Set the .BI File Cluster Size
UNIX DOS & OS/2	`proutil` *database-name* `-C truncate bi -bi` *kilobytes*
VMS	`PROGRESS/UTILITIES=TRUNCATE_BI/BI_CLUSTER_SIZE=`*integer database*

BTOS/CTOS	`PROGRESS Utilities`
	`Database Name` *database-name*
	`Utility Name -C` `truncate bi`
	`[Options]` `-bi` *kilobytes*

Specify the cluster size in kilobytes. The number must be a multiple of 16 in the range 16–262128 (16KB–256MB). The `.bi` file is organized into *clusters* on disk. At least four clusters are allocated at all times. As PROGRESS writes record before-images and notes to the `.bi` file, clusters fill up. Each time a cluster fills, all modified database buffers blocks (–B) are flushed to disk and either a new cluster is allocated, or the oldest one is reused. Raising the cluster size reduces the frequency of this substantial overhead. In general, the bigger the cluster size, the better the performance, but there are two significant drawbacks: (1) large clusters consume disk space, and (2) they can greatly increase the time required to do crash recovery when a database restarts.

6.1.1 USING THE procopy UTILITY

The previous section described how to use `procopy` to:

- Convert a void multi-volume database structure to a database you can use with PROGRESS by copying the PROGRESS empty database into a void multi-volume database.
- Copy a single-volume database to a multi-volume database.

You can also use `procopy` to:

- Copy a single-volume database to another single-volume database.
- Copy a multi-volume database to a single-volume database. Be sure you have sufficient file and disk space to perform the copy.
- Copy a multi-volume database to a multi-volume database.

NOTE: You cannot use procopy against a damaged database. You must first recover the database, then use the `procopy` utility.

6.1.2 Converting A Single-volume Database To A Multi-volume Database

The `procopy` utility also lets you convert an existing single-volume database to a multi-volume database, follow these steps:

1. Create a structure description file to define the appropriate data and before-image extents for the multi-volume database. Give this structure description file a different name than the database you want to convert.
2. Use the `prostrct` utility with the create option to create a void multi-volume database structure from the information in the structure description. Give this database the same name as the structure description file.
3. Use the `procopy` utility to copy the single-volume database to the void multi-volume database structure. The resulting database is then ready to use with PROGRESS.

If the *source* database is in use when you attempt to use `procopy`, `procopy` fails. You cannot use `procopy` against a crashed database; you must recover the database first.

You can use the `prostrct` utility with the `convert` option as an alternate method for converting a single-volume database to a multi-volume database. The `prostrct convert` command converts an existing single-volume database to a multi-volume database that is made up of exactly one data extent. You can then add additional data extents as desired.

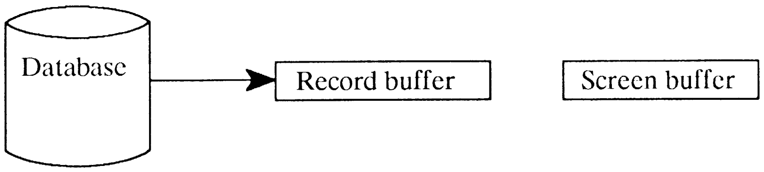
PROGRESS Language Reference Documentation Changes

(Version 6.2 and 6.2F)

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.
```

  - 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.

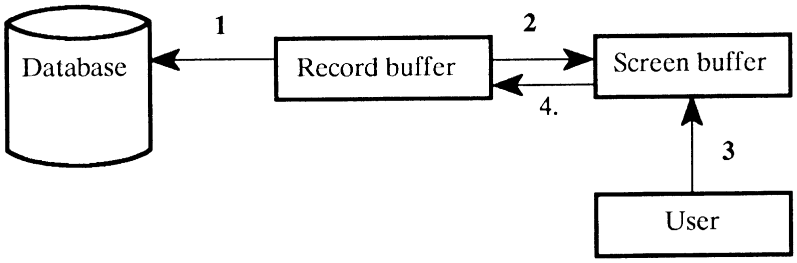
INSERT Statement

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                                                ]
      [ COLOR { [ DISPLAY ] color-phrase }  ...                 ]
      [       { 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.

PROGRESS Language Tutorial Chapter 16

(Version 6.2)

Chapter 16 Preparing Your Application for Use

You've made it! You're ready to put your application into production use. This chapter discusses these considerations and explains different options for handling each. It covers the following topics:

- Providing access to your application.
- Writing a gateway procedure.
- Backing up your database.
- Final steps.
- Using the Developer's Toolkit to distribute applications.

As you take that final step, there are some things you should consider:

- You must decide how your users will access your application.
- Depending on the type of application you are packaging, you may want to write a "gateway" procedure to tie the individual procedures together for your users.
- You must make provisions for backing up the application database.

You may find that you want to use the PROGRESS Developer's Toolkit to help in the last stages of development. This chapter describes some of the utilities that are available in the Developer's Toolkit and how they can help you when packaging your application.

16.1 PROVIDING ACCESS TO YOUR APPLICATION

There are different methods you can use to give users access to your application. You can choose to have the user do one of the following:

- You can let the user start PROGRESS and run procedures from the editor the way that you have done when developing the application.
- You can let the user start PROGRESS with a start-up option to run your application's main procedure. With this method, the user doesn't have to run your application from the PROGRESS editor.
- You can supply a file that automatically invokes PROGRESS with the start-up options that you define.
- You may want to set up a "captive user" by running the application automatically when the user logs onto the system.

The option you choose depends completely on the kind of access you want the user to have to both the operating system and to the PROGRESS editor. Table 16–1 summarizes operating system and editor access for each option. Note that this table applies to PROGRESS 4GL/RDBMS and PROGRESS Query/Run-Time only; PROGRESS Run-Time does not provide access to the PROGRESS editor.

Table 16–1: Operating System and Editor Access

Option	Access to the Operating System	Access to the PROGRESS Editor
Start PROGRESS and run procedures from the editor.	✓	✓
Start PROGRESS with the -p or /STARTUP option to automatically run your application.	✓	
Invoke a script, batch file, or command procedure that contains the start-up command and options.	✓	
Log onto the system and automatically run the application.		

Although in some cases the user does not have immediate access to the PROGRESS editor, you can provide access to the editor from your application. Later sections in this chapter explain how to provide that access.

16.1.1 Running PROGRESS to Access an Application

Using this option means that the user types the same command to start PROGRESS that you've been typing while developing your application. The user types one of the following commands:

Table 16–2: Command to Start PROGRESS

Operating System	Starting PROGRESS
UNIX	pro *database–name*
DOS & OS/2	pro *database–name*
VMS	PROGRESS *database–name*
BTOS/CTOS	Progress 4GL [Options] -1 *database–name*

NOTE: For startup information for networking clients, see the *System Administration I: Environments*.

Once the user has started PROGRESS and is in the editor, the next step is to run one or more PROGRESS procedures. For example, the user might run a procedure that displays a main menu for the application and remain in the application from that point on.

This approach might be useful in situations where you want users to have access to the editor for writing queries or other kinds of procedures. However, it has the disadvantage of appearing unfriendly to beginning users. It also requires users to enter any necessary startup options on the `pro` or `PROGRESS` command line.

16.1.2 Using a Startup Procedure to Run Your Application

Rather than have end users run your application from the PROGRESS editor, let them start both PROGRESS and your application with the same command line. Specifically, supply an application startup file that starts PROGRESS with the Startup Procedure (-p) option. The -p startup option lets you name a PROGRESS procedure to run automatically at start up.

Table 16-3: Command to Start PROGRESS and an Application

Operating System	Specifying a Startup Procedure
UNIX	pro *database-name* -p *procedure-name*
DOS & OS/2	pro *database-name* -p *procedure-name*
VMS	PROGRESS/STARTUP =*procedure-name database-name*
BTOS/CTOS	PROGRESS 4GL ⋮ [Start-Up Procedure -p] *procedure-name* ⋮ [Options] -1 *database-name*

In these commands, *database-name* is the name of the database you want the user to use and *procedure-name* is the name of the start-up procedure that runs once PROGRESS has started. This start-up procedure is usually a main menu, or gateway procedure, from which the user can run other application procedures. For example:

```
pro personnel -p mainmenu.p
```

This command starts PROGRESS using the personnel database and, instead of putting the user into the editor, it runs the mainmenu.p procedure. For complete information on startup parameters and parameter files, see Chapter 3 in *System Administration II: General.*

16.1.3 Providing a Script, Batch File, or Command Procedure

This method is similar to the method covered in the previous section, but instead of requiring the user to enter the command, you put that command in a script (UNIX), batch file (DOS and OS/2), command procedure (VMS), or submit file (BTOS/CTOS). Then all the user has to do is type the name of that file. PROGRESS then starts against a predetermined database and with a predetermined set of start-up options.

Here is a sample of such a batch file called SAMPLE.BAT. The cd command ensures that the user is in the appropriate working directory. The pro command starts PROGRESS using the personnel database, and runs the mainmenu.p start-up procedure.

```
sample.bat
```

```
cd \personnel
pro personnel -p mainmenu.p
```

The user can run the application by simply running the batch file **sample.**

16.1.4 Setting Up a "Captive" User

Sometimes you want a user to have access only to an application and not to the operating system or to the editor. If you are using DOS, you can add a line to the user's AUTOEXEC.BAT file that calls a batch file like the one you saw in the previous section. Similarly, if you are using OS/2, you can add a line to the user's CONFIG.SYS file. In this case, the user automatically runs the application when the computer is turned on. When the application exits, the user returns to the operating system.

On UNIX, BTOS/CTOS, and VMS, you can set up a captive user so that they cannot access the operating system.

To set up a captive user on UNIX, put the following line in the user's .profile file:

exec pro *database-name* -p *procedure-name*

If the user is running multi-user PROGRESS, use the mpro command in this line instead of the pro command. In addition, be sure to start the multi-user server before a user tries to start multi-user PROGRESS. You can incorporate this step into separate scripts or you can always start the server whenever a user wants to run multi-user PROGRESS (that is, precede the mpro command by the proserve command in the script). If the server is already running, trying to start it again will have no effect and does no harm.

Under UNIX, this method also logs the user out upon leaving PROGRESS.

If you are using VMS:

- Create a command procedure that starts PROGRESS and your application. This procedure should contain:

  - Definitions for certain logical names.

  - Use the ASSIGN or DEFINE command to assign SYS$INPUT to SYS$COMMAND so that VMS can look for commands from the command procedure and not from the terminal.
  - The PROGRESS command and any start-up qualifiers necessary for your application.

- Put the following line in the user's LOGIN.COM file:

  $@*file-name*.com

  In this command, *file-name* is the name of the command procedure that starts PROGRESS and your application.

If you are using BTOS/CTOS, put the following lines in the user's ".user" file:

```
:SignonVolume:volume-name
:SignonDirectory:directory-name
:SignonFileprefix:
:SignonPassword:
:SignonChainFile:[sys]<dlc>PROGRESS.RUN
PROGRESS SINGLE USER
database-name
procedure-name
:SignonExitFile:[sys]<sys>SIGNON.RUN
:ProgressEnv:[sys]<sys>progress.env
```

NOTE: If the user executes a BTOS/CTOS command that runs a submit file, the user can access commands by pressing Action-Finish while the submit file is executing. Or, the user can execute a BTOS/CTOS command without any parameters.

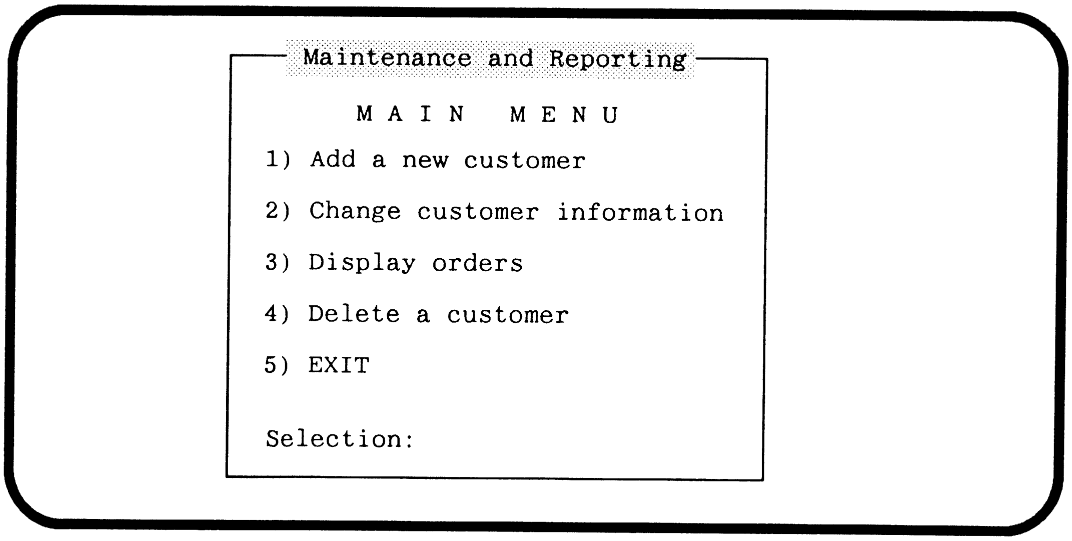
16.2 WRITING A STARTUP PROCEDURE

Regardless of which method you choose for user access to an application, you will need to write a "startup" procedure. A startup procedure is the first procedure a user runs (either by using the RUN command or automatically through the -p option or the /STARTUP qualifier) when entering PROGRESS. The startup procedure is usually an application main menu from which the user can run other application procedures.

A main menu procedure is much like any other menu procedure you have written. The only difference is that you need to think about the kind of access to the editor and to the operating system the procedure provides.

Here is one of the menu procedures used earlier in this manual:

t-csmenu.p
```
DEFINE VARIABLE Selection AS INTEGER FORMAT "9".
REPEAT:
  FORM
      SKIP(2) "    M A I N  M E N U              "
      SKIP(1) "  1) Add a new customer          "
      SKIP(1) "  2) Change customer information "
      SKIP(1) "  3) Display orders              "
      SKIP(1) "  4) Delete a customer           "
      SKIP(1) "  5) EXIT                        "
      WITH CENTERED
           TITLE "Maintenance and Reporting".
  UPDATE SKIP(2) SPACE(1) selection AUTO-RETURN
    WITH SIDE-LABELS.
  HIDE.
       IF selection EQ 1 THEN RUN t-adcust.p.
  ELSE IF selection EQ 2 THEN RUN t-chcust.p.
  ELSE IF selection EQ 3 THEN RUN t-itlist.p.
  ELSE IF selection EQ 4 THEN RUN t-delcus.p.
  ELSE IF selection EQ 5 THEN QUIT.
  ELSE MESSAGE "Incorrect selection - please try again".
END.
```


```
Maintenance and Reporting

      M A I N   M E N U

  1) Add a new customer

  2) Change customer information

  3) Display orders

  4) Delete a customer

  5) EXIT


  Selection:
```

Suppose this is the main menu for your application. Let's look at each of the different kinds of access you might want to give the user in terms of the ways the user might have started the application.

16.2.1 Controlling Access for the pro or PROGRESS Command Users

Suppose the user starts the application by:

1. Using the pro or mpro command (on DOS, OS/2 or UNIX), the PROGRESS or PROGRESS/MULTI_USER=LOGIN command (on VMS), or the PROGRESS Single User or PROGRESS Multi-user command (on BTOS/CTOS) to start PROGRESS.

2. Running the application main menu from the editor.

Once you have let a user access your application in this way, that user has access to the editor, operating system and to the menus of the application. Because this user already has fairly unlimited access, there is no real need to do anything special with your gateway procedure.

NOTE: Access is controlled by the password protecting files in BTOS/CTOS.

16.2.2 Controlling Access for -p, /STARTUP, and "Captive" Users

Now suppose you have decided to:

- Have DOS, OS/2 and UNIX users type the pro or mpro command along with the -p option and the name of your start-up menu procedure.

  Have BTOS/CTOS users type the PROGRESS Single User or PROGRESS Multi-User command then supply a startup procedure name at the "Startup Procedure" option.

  Have VMS users type the PROGRESS or PROGRESS/MULTI_USER=LOGIN command along with the /STARTUP qualifier and the name of your start-up menu procedure.

OR

- Have users type the name of a script, batch file, command procedure, or submit file.

OR

- Put users directly into the application when they log in.

In the first case, users already have access to the operating system and operating system access is not an issue for "captive" users. But, how can you control access to the editor?

First, let's assume you want the user to have access to the editor.

Go ahead and run the t-csmenu.p procedure. When the procedure prompts you for a selection, press END-ERROR (F4). The procedure puts you in the editor. Why? Remember the default PROGRESS action for the END-ERROR key:

- If you press END-ERROR (F4) on the first user interaction in a block, PROGRESS takes the default ENDKEY action which is UNDO, LEAVE.
- If you press END-ERROR (F4) after the first user interaction in a block, PROGRESS takes the default ERROR action which is UNDO, RETRY.

When you press END-ERROR (F4) in response to the selection prompt, you are on the first screen interaction of the block. So PROGRESS undoes the work you've done (if any), and leaves the block. Since there are no more statements in the procedure, it ends. PROGRESS returns you to the editor when the procedure you run from the editor ends or when the procedure named in a start-up option ends. Once in the editor, the user can use the RUN statement to access the application main menu again.

Now, suppose you don't want the user to have access to the editor. To accomplish this, simply add an ON ENDKEY UNDO, RETRY phrase to the REPEAT block of the gateway procedure.

t-csmen2.p

```
  DEFINE VARIABLE selection AS INTEGER FORMAT "9".

➤ REPEAT ON ENDKEY UNDO, RETRY:
    FORM
        SKIP(2) "     M A I N  M E N U              "
        SKIP(1) "  1) Add a new customer            "
        SKIP(1) "  2) Change customer information "
        SKIP(1) "  3) Display orders                "
        SKIP(1) "  4) Delete a customer             "
        SKIP(1) "  5) EXIT                          "
        WITH CENTERED
             TITLE "Maintenance and Reporting".
    UPDATE SKIP(2) SPACE(1) selection AUTO-RETURN
      WITH SIDE-LABELS.
    HIDE.
         IF selection EQ 1 THEN RUN t-adcust.p.
    ELSE IF selection EQ 2 THEN RUN t-chcust.p.
    ELSE IF selection EQ 3 THEN RUN t-itlist.p.
    ELSE IF selection EQ 4 THEN RUN t-delcus.p.
    ELSE IF selection EQ 5 THEN QUIT.
    ELSE MESSAGE "Incorrect selection - please try again".
  END.
```

Try running the t-csmen2.p procedure. Press END-ERROR (F4) in response to the selection prompt. Notice that the procedure no longer puts you in the editor. Instead, you remain at the selection prompt. By using the ON ENDKEY UNDO, RETRY phrase, you override the default action of UNDO, LEAVE.

Although you did not want users to have access to the editor, you did want them to be able to exit from the application. That is why the procedure includes an option (5) that runs the QUIT statement.

16.2.3 How Progress Handles the Stop Key

The way PROGRESS treats the STOP key depends on where in the application the user is when pressing that key:

- If the user is either directly in a start-up menu procedure or is running a procedure called by the start-up procedure, PROGRESS undoes the current transaction and restarts the start-up procedure. If you write your start-up procedure so that the user cannot get access to the editor with the END-ERROR (F4) key, then that user cannot access the editor with the STOP key either.

- If the user has left the application to work in the editor (assuming you allowed the user access to the editor), and then returned to running application procedures, pressing STOP returns the user to the editor.

If you run the t-csmen2.p procedure in the previous section, you can return to the editor by pressing STOP.

16.3 BACKING UP YOUR DATABASE

You should back up your application database as frequently as is necessary (usually on a daily basis) to ensure that you do not lose data due to a system failure. To back up a PROGRESS database, copy all of the files with the base name of your database (e.g. demo.*). You should never backup just the .db file. Always backup all related files as a group (.db, .bi, etc.).

NOTE: BE CAREFUL NOT TO BACKUP A DATABASE THAT IS IN USE. The exception is for performing online backups for multi-user databases. For more information about performing online backups, refer to Chapter 4 of the *System Administration II: General.*

A database is "in use" if a server is active, even if no users are currently using the database, or if the database is being used in single-user mode. Your backup scripts should use the proutil command described in Chapter 3 of the *System Administration II: General*. The proutil command should include the BUSY option to determine if the database is in use. If it is in use, do not perform the backup.

For extra protection, you should periodically create ASCII dumps of your databases using the PROGRESS Data Dictionary dump routine. You use this routine by choosing `Admin` from the Dictionary Main Menu and then choosing option D, Dump Data and Definitions, from the Admin menu. From the submenu, choose option D, to dump the Data Definitions (.df file). However, before dumping your database data, you should also dump the field and index definitions for each file in the database (first choose Admin from the Dictionary Main Menu, then choose option D from the Admin menu, and finally choose option F to dump the File Contents (.d files).

16.4 FINAL STEPS

Here is a checklist of the steps you should take when you are ready to put your application into use:

1. Choose a method for writing your start-up menu procedures.

2. Create a "basic" database. A basic database contains the schema for your application and perhaps some control data but is otherwise empty. You then store this basic database in a master directory so that you can make copies of it whenever necessary. To create a basic database:

    - From your development database, use option D, Dump Data and Definitions, from the Admin option of the Data Dictionary Main Menu. From the Dump Data and Definitions menu, choose option D to dump your data definitions. If there is control data in any files, use option F to dump the contents of those files. If you have stored any records in the _User file, choose option U to dump the contents of the _User file.

    - Create a new database from the empty database.

Table 16–4: Command to Create a Database

Operating System	To Copy The Empty Database
UNIX, DOS, and OS/2	prodb *database-name* empty
VMS	PROGRESS/CREATE *database-name* empty
BTOS/CTOS	PROGRESS Create Database New Database Name *database-name* Copy From Database Name empty

- In the new database (the "basic database"), use the L option, Load Data and Definitions, from the Admin option of the Data Dictionary menu. From the Load Data and Definitions menu, choose option D to load the dumped data definitions into the basic database. Use option F to load any data file contents, and option U to load any _User file contents.

3. Use the COMPILE SAVE statement to precompile your application procedures against the basic database.

4. Store the source versions of your application procedures, the object versions of your procedures, and the basic database in a separate directory and be sure to exercise control over changes to that directory. Also be sure to first test any changes in another directory before incorporating those changes in the master version.

   If you want to provide only object versions of procedures to users, you must be sure that you compile the procedures using a copy of the basic database that is identical to the one distributed with the application.

   When you distribute object files to users, you can place the files in a PROGRESS library. Placing object files in a library increases the overall performance of your application. To read more about placing object files in a library, see "Building Libraries with the Prolib Utility" in Chapter 4 of the *System Admininstration II: General.*

5. Whenever you need a fresh copy of the database, use the prodb command to copy the basic database from the master directory.

6. If you add or delete file, field, or index definitions in the basic database, you must recompile all procedures which access the file affected by the change against the modified database. Whenever you make these kinds of changes to the database, PROGRESS puts a new time stamp on the files that have changed. All procedures you compile against that database then contain that time stamp in their object files. The time

stamp of an object file must match the time stamp of the files referenced in the procedures against which you are trying to run that file.

If you want to provide Query/Run-Time or Run-Time users with the new database and the revised application, those users will have to dump the data from the old database and reload it into the new. This and other related considerations are covered in detail in the documentation that accompanies the Developer's Toolkit product.

7. If you make any other changes to the basic database, those changes are not reflected in procedures until you recompile those procedures against the modified basic database.

16.5 USING THE DEVELOPER'S TOOLKIT TO DISTRIBUTE APPLICATIONS

When you develop applications you plan to distribute to end users, you typically make several decisions: Do you want end users to be able to modify your application procedures and possibly add some of their own? Do you want end users to be able to access your database definitions and perhaps modify them?

Then there are the additional tools you have to supply along with your application: a facility to dump and reload the database (if the user has PROGRESS Query/Run-Time or PROGRESS Run-Time), batch files or scripts to start PROGRESS and access your applications, and so on.

The PROGRESS Developer's Toolkit consists of a complete set of tools to help you address these and other needs:

- dbrstrct Utility

  Use this utility to specify the kinds of access end users have to an application database.

- mkdump Utility

  You must supply a dump/reload facility to Query/Run-Time or Run-Time users. Use the mkdump Utility to create that facility.

- procomp Utility

  Use this utility to compile your procedures on a machine other than your development machine. By doing this compilation, users can run your application on a machine other than the machine on which you developed your application.

- tailor (UNIX only) Utility

You will need to make global changes to the values of certain variable definitions used in the Developer's Toolkit scripts. In addition, you may want to copy all scripts to /usr/bin. You run tailor to make these changes.

- xcode Utility

  Use this utility to encrypt source code for shipment to customer sites (where it can be compiled by a special version of the Developer's Toolkit compiler, procomp).

- Empty database

  You may want to make changes to the metaschema of the empty database supplied with PROGRESS. If so, you make those changes to the empty database supplied with the Developer's Toolkit.

- Templates of scripts to provide to users.

16.6 SUMMARY

This chapter showed how to pull all of your procedures together and present your application to your users. Specifically, it explained:

- Different methods for providing application access to end users.
- How to write start-up procedures.
- Database backup rules to remember.

This chapter also included a checklist of the steps you should take before putting your application into production use. The checklist is in the section called "Final Steps."