DB/C DX 101
Guide and Reference

January 2023

Table of Contents

DB/C Language Fundamental Concepts

This chapter covers fundamental concepts of the DB/C programming language. The information in this chapter is intended for the programmer who is new to DB/C.

Statement Structure

Programs consist of statements which are lines of text. All statements have the following general format:

label verb operands comment

The label is optional, but if it exists, it must start in the first column of the line. Blanks and tabs separate the label, verb, operands, and comment.

Here is an example of a complete DB/C program:

```
password char 8
         keyin *es, "Enter the password:", *eoff, password
         match "87654321" to password
         stop if not equal
         chain "menuprog"
```

In the first statement, the label name is **password**. This statement defines an eight-character variable named **password**. The following statements do not contain a label. The statement with the verb name **keyin**, simply called the keyin statement, waits for a password to be entered at the keyboard. ***es** is a control code that causes the screen to be erased and ***eoff** is a control code that causes characters entered on the keyboard to not be echoed to the screen. The match statement tests for a valid password. If the password is invalid, the stop statement causes execution to cease. If the password is valid, the chain statement ends the current program and starts the program named **menuprog**.

Variables

A variable is defined by a data definition statement. A data definition statement may be located at any place in a program. The only restriction is that a variable must be defined before any executable statements which use that variable.

The maximum number of characters that a variable name can contain is 31. The first character must be an alphabetic character (**A-Z**, **a-z**) or a dollar sign (**$**). The characters after the first character may be any alphanumeric character (**A-Z**, **a-z**, **0-9**), a dollar sign (**$**), a period (**.**), an underscore (**_**), or an "at" sign (@).

Three common types of variables are character variables, numeric variables, and file variables.

Character variables

A character variable contains a string of characters. A character variable is defined by a char or init statement. The char statement defines a character variable and initializes it with blanks. The init statement defines a character variable and initializes it with the characters specified.

Here are two examples of character variable definitions:

```
name     char 30
company  init "DB/C Software Company"
```

The first variable, **name**, can hold up to 30 characters. The second variable, **company**, can hold up to 21 characters. This is inferred from the length of its initial value: which is **DB/C Software Company**.

A character variable has several attributes associated with it. These attributes are: the physical length, the physical string, the form pointer, the length pointer, the logical string, and the logical length. The physical length refers to the largest number of characters that can be stored within a character variable. It is established when the variable is originally declared. For instance, if a variable is defined as char 8, then its physical length is 8.

The string of characters stored in a character variable is referred to as the physical string of the variable.

DB/C maintains two pointers with each character variable: the form pointer and the length pointer. Each pointer is a number that refers to a character position in the variable. For example, if the form pointer of a certain variable has a value of 9, then the form pointer refers to the character in the ninth position in the variable.

The value of the form pointer can never exceed the value of the length pointer. The value of the length pointer can never be greater than the physical length. The form pointer and the length pointer may have the value zero.

The form pointer and length pointer define the variable's logical string. The logical string of a character variable is the string of characters from the character pointed to by the form pointer through the character pointed to by the length pointer, inclusive. In other words, the form pointer points to the first character of the logical string and the length pointer points to the last character of the logical string. If the form pointer is zero, the logical string is considered to be null, even if the length pointer is non-zero.

The logical length of a character variable is the length of its logical string. If the form pointer is zero, then the logical length is zero regardless of the value of the length pointer. The logical length can be calculated by the following formula:

*logical length = length pointer - form point*er + 1

For example, if a variable is defined like this:

```
name      init "ABCDEFGHIJ"
```

The physical length of name is 10. The physical string contains the characters **ABCDEFGHIJ**. If the form pointer is 2 and the length pointer is 5, then the logical string is **BCDE** and the logical length is 4.

Numeric variables

A numeric variable contains a valid DB/C number. The characters that make up a valid DB/C number are always right justified in the variable and leading zeros are always suppressed. If the number is negative, a minus sign is stored on the left side of a number. The minus sign is always immediately to the left of the most significant digit, or to the left of the decimal point if there is no significant digit left of the decimal point.

Here are examples of valid and invalid numbers:

Valid: `"123"`   `"-123"`   `"   -123"`   `"-.05"`
Invalid: `"123 "` (trailing blank)   `"- 123"` (embedded blank)   `"123."` (no digit after `.`)
`"+123"` (`+` invalid)   `"1,234"` (`,` invalid)

Numeric variables may be defined by the num statement. The num statement defines the format of a numeric variable. The num statement can also be used to assign an initial value to a variable. To initialize a numeric variable, a numeric value enclosed in quotes is placed to the right of **`num`** verb. The size and format of the variable are inferred from the initial value.

Here are some examples of numeric data variable definitions:

```
n1        num 3
cost      num 3.2
n2        num "-66"
maxprice num "250.00"
```

The first variable, **n1**, is defined to hold three decimal digits. A minus sign takes the space of one digit. Therefore, **n1** can hold values ranging from -99 through 999. **cost** contains five digits and a decimal point. Unlike the minus sign, the position of the decimal point remains fixed. The decimal point takes up one character position. Data is always aligned around the decimal point.

cost can hold three digits to the left of the decimal point and two digits to the right. Therefore, **cost** can hold values ranging from -99.99 through 999.99.

n2 is defined as a **num 3** variable. This is inferred from the initial value assigned to the variable.

maxprice is defined as a **num 3.2** variable. This is inferred from the initial value assigned to the variable.

A numeric variable can be initialized larger than the length of its initial value by adding leading zeros or spaces to the number enclosed in quotes. In either case, the variable is initialized with leading blanks instead of zeros.

For example:

```
n3       num " 10"
n4       num "000001"
```

n3 is a **num 3** variable initialized to the value 10. **n4** is a **num 6** variable initialized to the value 1.

File variables

A logical file is defined by a file variable. To use an actual physical file in a program, the file must be linked to a file variable. Once the link is established, the file variable is used to refer to the file for all subsequent accesses. The link is established by the execution of an open statement. A file variable that is linked to a physical file is said to be open. When a file variable is open, data can be read from or written to the file. A file variable that is not linked to a file is said to be closed. When a file variable is closed, data cannot be read from or written to the file.

A direct file variable is used to allow access to a data file through the sequential or random access methods. It is defined by the file statement.

An indexed sequential file variable is used to allow access to an index file and its associated data file through the indexed sequential access method (ISAM). It is defined by the ifile statement.

An associative index file variable is used to allow access to an index file and its associated data file through the associative index access method (AIM). It is defined by the afile statement.

For example:

```
names    ifile fixed=80, keylength=8
```

This ifile statement defines **names** as a logical name for a file that can be accessed with the indexed sequential access method.

Flags

There are four flags that are used to control program execution. The flags have two states: set and clear. Most statements change the state of one or more flags, reflecting their results.

The flags affected depend on the statement. The flags are: **equal**, **less**, **over**, and **eos**. **over** is an abbreviation of "overflow". **eos** is an abbreviation of "end-of-string". **zero** is a synonym for **equal**.

The operation of certain statements (such as the goto statement) depends on the status of the flags.

The move Statement

The move statement is the basic assignment operation of the DB/C language. The move statement copies character variables, numeric variables, and literal values to character and numeric variables.

Here are some examples:

```
a        init "AAA"
b        init "BBBBB"
c        num 3.2
d        char 10
         move a to b
         move "-22" to c
         move c to d
         move "X" to b
```

The first operand in a move statement is called the source. The second operand is called the destination. Typically the source is moved to the destination, but several rules govern the move:

1. If the source is a character variable, then the characters in its logical string are moved.

2. If the destination is a character variable, then characters are moved into the destination variable starting at the first character. The destination's form pointer is set to one and length pointer is set to the number of characters moved. If no characters are moved, the destination form pointer and length pointer are both set to zero.

3. If the destination is numeric, then the source must be a valid number. If it is not, the numeric destination variable is not changed.

The move statement affects the flags. If both source and destination are numeric, then the equal, less, and over flags are affected. The equal flag is set if the value in the destination is zero; otherwise it is cleared. The less flag is set if the value in the destination is negative; otherwise it is cleared. The over flag is set if any digits or the minus sign was truncated on the left; otherwise it is cleared. If neither the source nor the destination is numeric, the eos flag is the only flag affected. The eos flag is set if characters were lost because they could not fit into the destination; otherwise it is cleared.

Character Variable Manipulation

The reset statement causes the form pointer of a character variable to be changed.

The cmove statement causes one character to be stored at the character position pointed to by a character variable's form pointer.

For example:

```
x        init "ABC"
         reset x to 2
         cmove "Z" to x
         reset x
```

The first reset statement sets the form pointer of **x** to 2. The cmove statement replaces the character **B** with a **Z**. The second reset sets the form pointer of **x** to 1.

The bump statement increments a character variable's form pointer.

The movefptr statement moves the value of a character variable's form pointer to a numeric variable.

The cmatch statement compares the form pointed characters of two variables. If the characters match, the equal flag is set. If the second operand is less than the first operand, the less flag is set. If the form pointer of either variable is zero,the eos flag is set.

The match statement is similar to the cmatch statement, except that the logical strings of two character variables are compared.

For example:

```
x        init "ABCDE"
n        num 5
         bump x by 2
         cmatch "A" to x
         movefptr x to n
         match "CDE" with x
```

In this example, the bump statement causes the form pointer of **x** to be incremented from 1 to 3. The cmatch statement clears the equal, less and eos flags. The movefptr statement moves 3 to the variable **n**. The match statement sets the equal flag.

The goto Statement

The goto statement causes program execution to continue at another location in the program. There are two types of goto statements: unconditional and conditional.

An unconditional goto statement causes execution to continue at the statement specified by the execution label.

With the conditional goto statement, program execution continues at the statement specified by the execution label only if the condition following the **if** is true. If the condition is false, program execution continues with the statement that follows the goto statement.

The clause following the **if** may be an expression, a flag, or a function key specification.

Here is an example of an unconditional goto statement:

```
         goto label1
         display "Processing"
label1   display "Done"
         stop
```

Execution continues at **label1** unconditionally and the first display statement does not execute.

Here is an example of a conditional goto statement:

```
         goto label1 if F1
         display "F1 not pressed"
         goto label2
label1   display "F1 pressed"
label2   stop
```

If the F1 key has been pressed before this code executes, then the condition in the first goto statement is satisfied (**if F1**). As a result, execution continues at **label1** and the message **F1 pressed** is displayed.

If the F1 key has not been pressed, then the condition in the first goto statement is not satisfied and execution continues with the next statement, which displays the message **F1 not pressed**.

The if, else, and endif Statements

The if, else, and endif statements conditionally control execution of subsequent statements. The if statement tests whether a condition is true or false. If it is true, execution continues with the next statement. Otherwise, execution continues after the else statement, or after the endif statement if an else statement does not exist.

Here is an example:

```
         if over
                  display "Over flag is set"
         endif
```

If the over flag is clear, the statement following the endif statement is the next statement executed. If the over flag is set, the condition specified in the if statement is true and the message **Over flag is set** is displayed.

Here is another example:

```
score    num 3
         keyin *es, "Enter your score:", score
         if (score = 100)
                  display "A perfect score"
         else if (score < 0 or score > 100)
                  display "An invalid score"
         else if (score >= 70)
                  display "A passing score"
         else
                  display "A failing score"
         endif
```

The display Statement

The display statement performs functions such as displaying characters on the computer screen, positioning the cursor, and setting display attributes.

The list of operands after the display verb defines the operations to be performed. The list of operands includes data elements (such as character variables, numeric variables, and literals) and control codes. The data elements contain the information to be displayed on the computer screen. The control codes modify the manner in which the data is displayed. All control codes begin with an asterisk.

Here is an example of the display operation:

```
          display *es, "This is an example"
```

The **`*es`** control code erases the screen and sets the cursor position to the upper left corner of the screen. Next, the string of characters
This is an example is displayed. After the characters are displayed, the cursor is positioned to the right of the last character displayed.

The cursor position designates the position on the display screen where the next character will be displayed. At times a flashing rectangle or underscore may appear at the cursor position. This rectangle or underscore is called the cursor.

The upper left corner of the screen is cursor position 1:1.

The following example displays **Hello** at the tenth position on the third line:

```
          display *p=10:3, "Hello"
```

When a character variable is encountered in the list of operands, the characters from the first physical character of the variable through the character pointed to by length pointer are displayed. Blanks are displayed for the number of characters that are between the length pointer and the maximum length of the variable. In this way, the number of characters displayed is equal to the maximum length of the character variable.

This example displays **XYZ** followed by two blanks:

```
x         init "ABCDE"
          move "XYZ" to x
          bump x
          display x
```

Note that even though the form pointer of **x** is 2, the first three characters are displayed followed by two blanks.

When a numeric variable or literal is encountered in the list of operands, its characters are displayed starting at the current cursor position.

The keyin Statement

The keyin statement is similar to the display statement, but it also accepts keystrokes that are typed on the keyboard.

Most display control codes work the same way with a keyin statement.

When a variable is encountered in a keyin operand list, the cursor appears at the current cursor position and the program waits for characters to be typed on the keyboard. As each character is typed, it is displayed and the cursor advances one position to the right.

When a character variable is encountered in the operand list, each character typed at the keyboard is placed into the variable starting at the first character position. When the ENTER key is pressed, the keyin operation for that variable completes. If more characters are entered than will fit in the character variable, only those characters that will fit are accepted. If the user presses the ENTER key without typing any characters, then the form pointer and length pointer of the variable are set to zero.

When a numeric variable is encountered in the operand list, only characters that comprise a valid DB/C number that will fit in the variable may be entered. When the ENTER key is pressed, the keyin operation for that variable completes and the characters entered are moved to the numeric variable. The format of the numeric variable is preserved. For example, if the user attempts to enter 123.456 into a numeric variable with a 4.2 format, the number 123.45 is moved to the variable with one leading space. The 6 is lost. If the user presses the ENTER key without typing any characters, the numeric variable is set to zero.

Here is an example that uses both the display and keyin statements:

```
name     char 30
         display *es, "Enter your first name: "
         keyin *p=24:1, name
         display *p=1:3, "Hello, ", name
```

The screen would look like this:

```
Enter your first name: Fred
Hello, Fred
```

File Handling

DB/C supports three methods of file access: the direct access method (random or sequential), the indexed sequential access method (ISAM), and the associative index method (AIM).

A file variable is used with the direct access method. An ifile variable is used with ISAM. An afile variable is used with AIM.

A file is a set of data records that exists outside of the DB/C program. Before a program can process data that is stored in a file, it is necessary to connect a file variable to the file.

One file is used with the direct access method: a data file. Two files are used with the ISAM or AIM access methods: a data file and an index file. A data file contains records of data. An index file contains the information necessary to find records in a data file. An index file also contains the name of the corresponding data file. The index file is called an ISAM index file or an AIM index file, depending on the access method used.

Direct access

Sequential access provides a straight forward way to read and write files. Sequential access simply means that records are written one after another to the data file. Records are read from the data file in the same sequence that they were written to the file - one record after another. With sequential access, records may be accessed consecutively, either forward or backward. Sequential files are used when there is no requirement to access the data by record number or by key.

Random access allows reading or writing of fixed-length records by record number. Data files with varying length records can not be accessed randomly. DB/C assigns a non-negative integer number to each record in the file. A record can then be accessed by specifying its assigned number. Record numbering begins with zero. This means that the first record in the file is record number zero, the second record is record number one, etc.

ISAM

The indexed sequential access method (ISAM) allows a program to locate specific records based on a key value. This method uses a data file and an index file. The data file contains records that have key values. The index file contains the name of the data file, the key values, and the file positions necessary to associate the keys with the records in the data file.

For example, assume there is a data file that contains four million records. Each record contains the name, state of birth, and social security number of a person. The records are sorted alphabetically by the person's last name.

If you need to find the record for someone by the name of Moe Howard, you can find it easily because the data is sorted alphabetically by last name. However, the task becomes much more difficult if you need to find the record for the person whose social security number is 000-536-8912. You would have to read every record sequentially until you find the right one. On the average, you would have to read two million records. Indexing the file by social security number is a better solution.

For every record in the data file, a record is written to another file - the index. Each of these index records contains the social security number, also known as the key data. Each index record also contains a pointer that indicates where in the data file the record with that social security number is located. If this index file is arranged by social security number, then you have the means to find data records quickly. You can look up a key in the index in order to establish the exact position of the desired record within the data file.

ISAM does all this for you. ISAM gives you the ability to read data records in the same order as the index file. In this example, you can create a report that has every record from the data file printed in social security number order.

Because the index files are logically separate files, more than one index file can be associated with a single data file. In this example, there could be a state of birth index along with the social security number index. Both indexes would refer to the same data file.

When a change is made to a key field in a data file record, any related index files will be inaccurate. All indexes must be updated whenever a key field is changed in the data file.

AIM

The associative index method (AIM) is used to find records when only partial key information is known. AIM is used to read records that meet a specific set of criteria. As with ISAM, the data file contains records that have key values. The associative index file contains the name of the data file and the key information that is used to retrieve records from the data file.

For example, assume that you are designing a system for a customer service department. The customer file contains 200,000 records. Customers call the department frequently to inquire about the status of their orders. Unfortunately, most customers do not know their customer ID numbers. AIM can be used in the customer service system to quickly determine the ID number for a particular customer.

When a customer calls, he is asked for his name and zip code. Suppose Sue Smith calls from the 01243 zip code and does not know her customer ID number. If you only had ISAM at your disposal, you could search for all the customers in the 01243 ZIP code area, but that would likely yield too many records to be of any use. Similarly, searching under the name Smith would not be helpful.

AIM can narrow the search and help you find Sue Smith in the customer file. With AIM, you can construct a search that stipulates that the character strings Sue and Smith must both exist somewhere in the name field, and that ZIP code must be 01243.

Such a search could yield the following records:

Customer ID	Customer Name	Customer Zip Code
10036	Sue Smith	1243
91654	Sue's BlackSmithing	1243
62345	Smith Suede Company	1243
12634	Sue Smithers	1243

The user viewing these records can quickly determine that customer ID 10036 is the correct customer. Note that the search arguments Sue and Smith appear in every name field, but not in any particular location.

Records are selected by specifying a match pattern. This match pattern can contain a string of characters in one or more fields of a record. Only those records that satisfy the match pattern are retrieved. The data contained in the match pattern is called the key data. The key data is used to match the data in the record fields.

Key data may contain wild card characters. The wild card character can be used to specify a more flexible search pattern. The positions in the key data corresponding to the positions of the wild card characters may contain any character.

For instance, if you are looking for the name Carlson, but you are not sure of its spelling (it might be Karlson or Carlsen), then you could employ wild card characters. The default wild card character is the question mark. So, to conduct a general search that would result in every possible spelling, the key data could be: **?arls?n**.

Notice how the match pattern contains two wild cards. When the AIM read is executed, all names that contain the character string **arls** and have an **n** in the last character position will match the key data, regardless of the values of the first and sixth characters.

Unlike ISAM, AIM does not allow you to specify the order in which records are retrieved.

The print Statement

The print statement sends data to a printer or to a print file. The print statement is similar to the display statement. The list of operands includes character variables, numeric variables, literals, and control codes.

The print statement places characters into a print line buffer. The print line is then transferred to the printer or to the print file. The print position determines the horizontal placement of each character within the print line buffer. The print position starts at position one (the far left character position) in the print line. After each character is printed, the print position is set to the position to the right of the last character printed.

When a character variable is in the operand list, characters are printed from the first physical character through the character pointed to by the length pointer. A blank is printed for each character position after the length pointer through the physical length of the variable. If the form pointer of the variable is zero, then blanks are printed for the full physical length of the variable. Printing starts at the current print position.

Here is an example of print used with a character variable:

```
string   char 20
         move "ABCDEFGH" to string
         print string
```

ABCDEFGH is printed with 12 trailing blanks. The variable **string** is defined to be of length 20, with form pointer of value one and length pointer of value eight. Therefore, 12 trailing blanks are printed for each character in **string** after the length pointer.

Numeric variables and literals are handled the same way by the print statement. When a numeric variable or literal is in the operand list, all characters are printed, including any leading blanks. Printing starts at the current print position.

Here is another example:

```
         print *f, "Report of sales commissions":
                  *n, *n:
                  "Salesperson":
                  *tab=25, "Amount"
```

The colon is used to continue the print statement on multiple program lines. The lines that follow the first line are indented to increase readability.

***f** is the form feed control code. It causes the printer to advance to the top of the next page and the print position to be set to one.

***n** is the new line control code. It causes the printer to advance one line and the print position to be set to one.

***tab=25** is a tab control code. It is used to set the current print position within the print line. The ***tab=25** in the fourth program line sets the current print position to the 25th column of the current print line.

Using Advanced DB/C Language Features

Primary and Secondary Program Modules

A compiled DB/C program is called a module. A module can be primary or secondary. A module is a primary module if it is the first program started by the DB/C run-time or if it is loaded and executed using the chain statement. A module is a secondary module if it is loaded by the loadmod statement or the ploadmod statement.

Program execution begins with the first statement of a primary module and can continue into secondary modules by execution of goto, call, branch, perform, or user-defined verb statement.

Here is an example:

Program 1:

```
. This is the program started using the DBC command
         chain "first"
```

Program 2:

```
. This is the module named first
xlabel   external
         loadmod "second"
         goto xlabel
```

Program 3:

```
. This is the secondary module named second.txt
xlabel   routine
         display "Secondary module is executing"
         stop
         endroutine
```

The chain statement in Program 1 starts execution of a new primary module named **first**. The first executable statement of the **first** program, the loadmod statement, loads a secondary module named **second**. The goto statement causes execution to be transferred to the label **xlabel** in **second**. The message **Secondary module is executing** is displayed and then the stop statement is executed.

Unlike the primary module, there can be multiple copies of a secondary module. Each copy is loaded by a separate loadmod statement. Each copy of a secondary module is called an instance.

The loadmod statement is used to create and to switch between instances of secondary modules.

For example:

```
         loadmod "second<one>"
         loadmod "second<two>"
```

These two statements load two copies of the **second** module: **<one>** and **<two>**. The **<two>** instance is the current instance. The current instance is the copy that becomes active by using a goto statement, call statement, branch statement, perform statement, or user-defined verb statement to transfer execution into the **second** module.

The unload statement is used to destroy one or all instances of a secondary module. The unload statement can also unload all secondary modules.

Execution of a chain statement by the primary module or a secondary module unloads the previous primary module and all secondary modules. The new primary module specified with the chain statement then begins execution.

A module is a permanent secondary module if it is loaded by the ploadmod statement. The unload and chain statements do not affect permanent secondary modules.

External Labels

Each program execution label is local to the module in which it is compiled unless the label has been specifically defined as a global (or external) label. A local label is unknown to other modules. A global label is visible to other modules.

A label definition is made global by the routine statement.

The external statement causes a label reference to refer to a global label defined in another module.

Here is an example of the loadmod statement used with multiple instances:

Program 1:

```
. This is the mod1 module
lab1     external
lab2     external
lab3     external
         loadmod "mod2<first>"
         call lab1
         loadmod "mod2<second>"
         call lab2
         loadmod "mod2<first>"
         call lab3
         unload "mod2<second>"
         unload
         stop
```

Program 2:

```
. This is the mod2 module
var1     char 1
lab1     routine
         move "X" to var1
         return
         endroutine
lab2     routine
         move "Y" to var1
         return
         endroutine
lab3 routine
         display var1
         return
         endroutine
```

X is displayed as a result of executing the **mod1** program.

In the **mod1** program, **lab1**, **lab2**, and **lab3** are defined as external labels be cause they are not found in the **mod1** program.

The first loadmod statement loads the program area and the data area of the **mod2** program. Notice that the name of the in stance is **<first>**. The **lab1** routine is called and execution continues in the secondary module. The letter **X** is moved to the variable **var1** in the data area of the **<first>** module instance.

Execution returns to the **mod1** program. The second loadmod statement loads the **mod2** program again, but notice that the name of the instance is **<second>**. A new copy of the data area is created. The **lab2** routine then is called and execution continues in the secondary module. The letter **Y** is moved to **var1** in the data area of the **<second>** module instance.

Execution returns to the **mod1** program. The third loadmod statement causes **<first>** to be made the current instance of the **mod2** program. The **lab3** routine is called and execution continues in the secondary module. The variable **var1** from the data area of the **<first>** module instance is displayed. Its value is **X**.

Execution returns to the **mod1** program. The first unload statement unloads the instance of the **mod2** program called **<second>**. The second unload statement unloads all currently loaded secondary modules.

Address Variables

An address variable contains a pointer to another variable of the same type. After an address variable is assigned a value (that is, an address of another variable), then the address variable may be used interchangeably with the variable to which it points. In other words, references made to an address variable work in the same manner as references made to the variable pointed to by the address variable.

An address variable is defined like any other variable, except it has an **@** as its operand.

This example defines a character address variable:

```
var1     char @
```

Address variables are assigned pointer values by several different operations, including the moveadr statement and the call statement with parameters.

The moveadr statement assigns the address of a specified source variable to an address variable.

Here is an example that uses the moveadr statement:

```
var1     char @
var2     init "Hello"
         moveadr var2 to var1
         display var1
```

Hello is displayed.

Access to variables is usually local to the current instance of a module. An exception is referring to a variable through an address variable. Address variables may refer to variables in any instance of any module.

Access across modules can be initiated by the call statement with parameters.

Here is an example using the call statement with parameters:

Program 1:

```
. This module is mod1
num1     num "1"
num2     num "2"
exlabel  external
         loadmod "mod2"
         call exlabel using num1, num2
         display "num1=", num1, " num2=",num2
```

Program 2:

```
. This secondary module is mod2
anum1    num @
anum2    num @
exlabel  routine anum1, anum2
         display "anum1=", anum1, " anum2=", anum2
         add anum1 to anum2
         return
         endroutine
```

When a routine statement is called by a call statement with parameters, the address of each variable in the call list is moved into the corresponding address variable in the routine list. An implicit moveadr operation is performed on each parameter.

In **mod1**, **exlabel** is defined as an external label. It is used by the call statement. The addresses of the variables **num1** and **num2** are moved into the address variables **anum1** and **anum2**. The following is displayed on the screen:

```
anum1=1 anum2=2
num1=1 num2=3
```

There is a special type of address variable called a typeless address variable. It is defined like this:

```
x        var @
```

A typeless address variable contains a pointer to any other non-address variable. The moveadr statement is used to load and store pointers in typeless variables.

Here is an example:

```
mmsg     init "Hello"
x        var @
y        char @
         moveadr msg to x
         moveadr x to y
         display y
```

`Hello` is displayed.

Local Variable Scope

A variable defined within the scope of routine/endroutine statements is called a local variable. A local variable is not usable after the scope-ending endroutine statement is encountered.

Here is an example of the definition and use of a local variable:

```
         goto start
r        routine parm1
parm1    char @
var1     init "def"
         display parm1, var1
         return
         endroutine
var1     init "abc"
start    call r using var1
         stop
```

In the example, the first **var1** variable is only valid until the endroutine statement. An attempt to display it after the start label would result in an undefined variable error during compilation. The program displays **abcdef**.

There is one other thing to notice about this program. There is one exception to the general rule that all variables must be declared before they are used. The exception is for parameters contained in a routine or lroutine statement. In this case the parameter declaration may immediately follow the routine or lroutine statement. The parameter **parm1** in the example shows the usefulness of this exception.

User-Defined Verbs

The verb statement defines a user-defined verb. A user-defined verb may be used just like a regular DB/C verb, except it executes an implicit call operation with parameters. The call is to a label that is the same as the name of the user-defined verb.

For example:

```
errmsg   verb #cvarlit
password char 8
         loop
                 keyin *es, "Enter password:", *eoff, password
                 if (lengthptr password < 8)
                         errmsg "Password must be 8 characters long"
                 else if (password < > "plato!!!")
                         errmsg "Invalid password"
                 else
                         break
                 endif
         repeat
         chain "menu"
errmsg   lroutine msg
msg      char @
work1    char 1
         display *p=1:10, "*** ERROR ***", *p=1:11, msg
         keyin *p=1:12, "Tap ENTER to continue", work1
         return
         endroutine
```

The user-defined verb, **errmsg**, is defined by the verb statement. The **#cvarlit** operand means that one, and only one, character variable or literal operand is required. The **errmsg** verb can then be used like any other verb. It is used twice in the example.

There are three general classes of operands allowed with user-defined verbs. These operand classes are: positional, non-positional, and keyword.

In the user-defined verb definition, the format of a positional operand is:

*type*

In the user-defined verb definition, the format of a non-positional operand is:

= *type*

In the user-defined verb definition, the format of a keyword operand is one of:

keyword
keyword **=** *type*
keyword **=** *type* **:** *type*
keyword **=** *type* **:** *type* **:** *type*
keyword **=** *type* **:** *type* **:** *type* **:** *type*
keyword **=** *type* **:** *type* **:** *type* **:** *type* **:** *type*

In the user-defined verb definition, positional operands precede non-positional and keyword operands. There may only be one non-positional operand, but there may be zero, one, or more of the keyword or positional operands.

The positional operand defines the syntax of the user-defined verb to require an operand of matching type in that position. The order of the operands with the verb must match the order of the positional operands in the user-defined verb definition.

The non-positional operand defines the type or types of operands that may be found in a comma delimited list after the required (positional) operands. The key word operands define the keyword operands and values that may be found in the same comma delimited list as non-positional operands.

Positional operands become the parameters of the call with parameters statement that is implicitly created by a user defined verb. Non-positional and key word operands are accessed by getparm, loadparm and resetparm statements.

This example shows the use of non-positional and keyword operands:

```
doprint  verb newline, p=nvarlit, =varlit
         doprint "line1", p=50, "at 50", newline, "line2", newline

doprint  lroutine
keyword  char @
var1     var @
char1    char @
num1     num @
vartype  num 2
         loop
                  getparm keyword, var1
                  return if over
                  if (formptr (keyword) = 0)
                           type var1, vartype
                           if (vartype=1)
                                    moveadr var1 to char1
                                    print char1;
                           else
                                    moveadr var1 to num1
                                    print num1;
                           endif
                  else if (keyword = "NEWLINE")
                           print *n;
                  else if (keyword = "P")
                           moveadr var1 to num1
                           print *tab = num1;
                  else
                           display "ERROR"
                           stop
                  endif
                  repeat
         endroutine
```

In the **doprint** user-defined verb definition, **NEWLINE** and **P** are key word operands. **NEWLINE** has no values associated with it. **P** has one which must be a numeric variable or literal. The non-positional operand may be a character variable, a numeric variable, or a literal.

The **doprint** routine consists of a large loop/repeat. Each iteration of this loop processes one parameter from the calling list. Each parameter is returned by the getparm statement. When the getparm statement sets the over flag, the end of the list of keywords and non-positional operands has been reached.

Object-Oriented Programming

The object-oriented features of DB/C are implemented with several different statements including class, endclass, make, destroy and the object data definition statements.

The class statement is used to define a class. A class definition looks like this:

classname **`class definition, parent=`***classname***`, make=`***routinelabel***`, destroy=`***routinelabel*

`endclass`

where the **parent**, **make** and **destroy** operands are optional. The statements between the class and endclass statements define the class. Variables that are outside of a routine, but inside of a class are called class variables. If the operand field of a class variable starts with a single **&** character, it is an inheritable class variable. If the operand field of a class variable starts with **&&**, it is an inherited variable definition. Inherited variables do not need to contain variable length information, format information or correct array size specification. All other information must be specified.

When a class is to be used by a make statement, or implicitly by execution of a user defined verb statement execution, the class must be defined with a class statement that does not contain the definition keyword. Here is the syntax:

classname **`class module=`***charlit*

If the class definition statement for the class being declared is contained in the same compiled module, then the **module=***charlit* operand is not specified. If the class definition statement is in another module, then *charlit* specifies the name of the module in which it is found.

Methods are routines that are contained inside a class definition. When a method is to be used by a call statement that exists outside the class definition the routine is defined in, it must be defined with a method statement, like this:

methodname **`method`**

An object is a data type (char and ifile are other data types). An object variable is defined like this:

objectname **`object`**

The make and destroy statements instantiate (create) and destroy an object. Here is their syntax:

`make` *objectname prep classname* **`with parmlist`**

`destroy` *objectname*

The **with** *parmlist* operand is optional.

There are two ways to call a method defined in a class—with a call statement, or with a user defined verb. Here is the syntax for calling a method with a call statement:

`call` *methodname*:*objectvariable* **`with`** *parmlist*

The **with** *parmlist* operand is optional.

Here is a small example of how to define and use a class. This example is contained in a single source file.

```
. main program
person class
showperson method
person1 object
         display *es, "Create and destroy person demonstration"
         make person1 from person
         call showperson:person1
         destroy person1
         stop

. definition of person class
person class definition, make=newperson
name char 30
birthdate char 6

newperson routine
         keyin "Enter name: ", name, *n, "Enter date of birth: ", birthdate
         return
         endroutine

showperson routine
         display "Name: ", name, "Date of birth: ", birthdate
         return
         endroutine

endclass
```

In the example, **person1** is the name of the variable that will contain an object of the class person. An object variable can contain an object of any class. The **person** class object is created by the make statement. When we say created, we mean that a set of data variables of the class is created and its reference is stored in the object variable. This is very similar to the DB/C concept of creating an instance of a loadmod. In our example, the two variables, **name** and **birthdate**, are created and the reference to their data area is stored in **person1**. The destroy statement removes the object (the data variables) and makes any references to them (such as the reference in **person1**) invalid.

When the object is created, a special routine is called to help create the object. In our example, this routine is called **newperson**. It is special because it is specified as the make method on the class definition statement. Note the term "method". This is the object-oriented term for a routine that is contained in a class. In our example, the **person** class has two methods, **newperson** and **showperson**. Note that in our main program, we only had to declare the **showperson** method because it is the only method explicitly used. The **newperson** method is called implicitly by the make statement. The make and destroy methods are optional for a class. In our example, we don't have a destroy method.

The call to the **showperson** method is similar to a regular call statement except that the routine name is followed by the object being referenced. A colon separates the method and the object variable. Note that there could be more than one class containing a **showperson** method. At run-time, the class of the object referred to by **person1** is used to determine which **showperson** method is called. This is called polymorphism.

The second way of calling a method is by defining the method as a user-defined verb. The method statement is not used. When a user-defined verb defines a method, the first operand of the prototype list on the verb statement must. be one of these forms:

`!make`
`!make(`*classname***`)`**
`!method`
`!destroy`
`!transient(`*classname***`)`**

The **!make** forms cause an object to be instantiated (an implicit make statement occurs before the method is called). The **!destroy** form causes an implicit destroy to occur on the object after the method is called. The **!transient** form causes a temporary, unnamed object variable to be instantiated before the method is called and then destroyed after the method returns.

The syntax for using the verb statement is similar to the syntax when using using a non-object oriented user-defined verb. For a **!transient** user-defined form the syntax is the same. For the **!make** form without the class name, the first operand after the verb is of this form:

object-variable **(***classname***)**

All other forms, including the **!make** form with the class name, require the first operand to be:

object-variable

Here are some examples:

```
person   class
newperson verb !make(person)
new      verb !make
showinfo verb !method
kill     verb !destroy
showhello verb !transient(person)
person1  object
         newperson person1
         kill person1
         new person1(person)
         showinfo person1
         kill person1
         showhello
```

Note that polymorphism isn't possible with the **!make(***classname***)** and **!transient(***classname***)** forms.

Here is an example of using the DB/C user verb syntax to call a method. In our example, after the **showperson** method statement in the main program, add the following line:

```
showperson verb
```

We can separate the class definition from the program that uses a class. In our example, let's assume the main program will be in a source file called **main.txt** and the person class source code will be in the file **class1.txt**.

The only change required is to change the class statement in the main program like this:

```
person   class module="class1"
```

The **module** parameter defines the name of the file with a **.dbc** extension that contains the code which defines the **person** class. Note the default extension is **.dbc**, just like for chain and loadmod. The file **class1.dbc** is not a library, it is just a normal **.dbc** file created by DB/C language compiler. Multiple classes can be defined in a single source file. Classes cannot be nested.

Inheritance is the most important object-oriented feature. As an example, let's assume we want to create a new class called **employee** that inherits all of the features of the **person** class and adds some. The **employee** class definition will be in a source file named **employee.txt**. Here is how we define it, assuming that the **person** class is in a file named **class1.dbc**:

```
. definition of employee class
person   class module="class1"
showperson method
employee class definition, make=newemployee, parent=person
salary num 6.2

newemployee routine
         keyin "Enter salary: ", salary
         return
         endroutine

showemployee routine
thisobj object @
getobject thisobj
         call showperson:thisobj
         display "Salary; ", salary
         return
         endroutine
         endclass
```

The main program looks like this:

```
. main program
employee class module="employee"
showemployee method
employee1 object
         display *es, "Create and destroy employee demonstration"
         make employee1 from employee
         call showemployee:employee1
         destroy employee1
         stop
```

Notice that the main program doesn't even know of the existence of a **person** class. When the make statement executes, an **employee** class object is created. The **parent** operand of the **employee** class definition statement specifies that a **person** object is also to be created and its variables are "merged" with the variables defined in the **employee** class. After the variables are created, the **person** class make method is called, after which the **employee** class make method is called. In the **showemployee** method, the getobject statement is used to set the **thisobj** variable to contain a pointer to the object variable for this instance of this class. **thisobj** is then used in the call to the **showperson** method which is actually contained in the parent class (that is the **person** class).

Variables can also be inherited. Inheritable class variables have a single **&** character as the first character of the operand field on their declaration. Inherited class variables have **&&** as the first characters of their operand field. In addition, because they are fully defined in the ancestor class, size information is not needed on inherited variables. Size information is the field size and the number of entries in each dimension of an array. Other type information is required. Please note that inheriting variables from an ancestor is optional. You only need to declare those variables in the new class that must be accessed directly.

Here are some examples of inherited variable declarations. In the ancestor, the base declarations are:

```
numvar   num &5.2
carray2  char &5[10,10]
arrayofptr num &[25]@
ptrtoarray num &@[]
file     ifile &keylen=20, fix=200
```

In the child class, the inheriting declarations are:

```
numvar   num &&
carray2  char &&[,]
arrayofptr num &&[]@
ptrtoarray num &&@[]
file     ifile &&
```

Here is an example using the **employee** class:

```
. definition of employee class
person   class module="class1"
employee class definition, make=newemployee, parent=person
salary   num 6.2
name     char &&
birthdate char &&

newemployee routine
         keyin "Enter salary: ", salary
         return
         endroutine

showemployee routine
         display "Name: ", name:
                  "Date of birth: ", birthdate:
                  "Salary: ", salary
         return
         endroutine
         endclass
```

Here is an example of a transient user-defined verb:

```
. this is the class definition code
message  class definition
errormsg routine text
text     char @
         display *hd, "Error occurred: ", text, w=5, *hd, *el;
         return
         endroutine
         endclass

. this is the main program
message  class module="message"
errormsg verb !transient(message), #cvarlit
         display *es, "Error message test program"
         errormsg "Unable to find file"
         stop
```

When the **errormsg** verb is executed, an unnamed object of class **message** is created, its make method is called (in this example, there is none), and the method whose name is the same is the user verb is called. When that method returns, the destroy method is called (in this example, there is none), the unnamed object is destroyed and the program execution returns to the calling program. This is quite convenient for execution of routines that do not need to have variables persist beyond their return.

Writing GUI Programs

This chapter is a guide to writing graphical user interface (GUI) programs.

Variables

Four special types of variables are used in GUI programs. They are: device variables, resource variables, queue variables, and image variables.

Device variable

A device variable is defined by the device statement. In GUI programs, windows and timers are represented by device variables.

Resource variable

A resource variable is defined by the resource statement. In GUI programs, menus, dialogs, icons, and toolbars are represented by resource variables.

Queue variable

A queue variable is defined by the queue statement. In GUI programs, a queue variable may be linked to a window, timer, menu, or dialog. Messages that result from user actions (like mouse movements and menu selections) are added to the queue.

Image variable

An image variable is defined by the image statement. An image variable contains a rectangular array of pixels that constitute a picture or drawing. Image variables may be shown in windows.

Devices

A device variable represents a general purpose object. In GUI programming, a window is represented by a device variable. The statements that operate on device variables are: open, prepare, close, link, unlink, change, query, load, and store.

The following prepare statement creates a window and displays it on the screen. For example:

```
win1     device
         prepare win1, "window=window01, size=100:100, pos=10:10"
```

win1 is the device variable that is used to refer to the window named **window01**. **size** specifies the horizontal and vertical size (in pixels) of the window. **pos** defines the horizontal and vertical position (in pixels) of the window relative to the upper left corner of the screen.

The close statement destroys a window and erases it from the screen. For example:

```
         close win1
```

Resources

Resources are objects that are manipulated and linked with windows. The statements that operate on resources are: open, prepare, close, show, hide, link, unlink, query, and change.

Here is an example that demonstrates window and resource creation:

```
dialog   resource
window   device
dlgdef   init "dialog=dbox, size=100:150, h=40, v=80,":
                 "defpushbutton=102:OK:20:10, h=10, v=30, edit=101::30"
         prepare window, "window=window01, size=100:100, pos=10:10"
         prepare dialog, dlgdef
         show dialog, window
```

The first prepare statement creates the window for this program. The second prepare statement creates the dialog resource named **dialog**.

The first operand in **dlgdef** is **dialog=dbox**. **dbox** is the name of the dialog.

The next operand is **size=100:150**. This operand specifies the size of the dialog box. The first value, 100, is the horizontal size. The second value, 150, is the vertical size. These values are in screen pixels.

The next operand is **h=40**. This operand specifies the current horizontal position within the dialog box. The 40 is the horizontal position in screen pixels. The **v=80** operand designates the current vertical position within the dialog box.

defpushbutton=102:OK:20:10 specifies the default push button control. A default push button is the push button that is activated when it is clicked on or when the enter or return key is pressed. 102 is the item number of the push button. **`OK`** is the text string that will be displayed in the push button. 20 is the horizontal size of the push button in screen pixels. 10 is the vertical size of the push button in screen pixels. The upper left corner of the push button control will be placed at the current position.

The next operands, **h=10** and **v=30**, set the new current position within the dialog box.

The final operand is **edit=101::30**. This specifies an edit field control. 101 is the edit field control item number. The second operand specifies the text string that will be displayed in the edit field. In this case, no text string is specified, so the edit field control is initially empty. 30 is the horizontal size of the edit field in screen pixels. The upper left corner of the edit field control will be placed at the current position.

The show statement displays the dialog box in the center of the screen.

Queues

A device or resource is linked to a queue variable by the link statement. This allows windows, menus, and control boxes to send messages to the program through queues.

The get statement retrieves messages from a queue in the order that messages were placed in the queue (that is, first in, first out).

The wait statement causes program execution to be suspended until a message is available. Execution continues with the statement after the wait statement when a message is available.

Here is an example that demonstrates the use of the get and wait statements:

```
q1       queue size=20
msg      char 20
         loop
                 wait q1
                 get q1; msg
         repeat
```

q1 is a queue variable that can hold 20-character messages. At the start of the loop, the program pauses until **q1** is non-empty. When a message is available, it will be moved to the variable **msg**. By default, **q1** can hold a maximum of 32 entries.

Here is a complete example:

```
dialog   resource
window   device
msgqueue queue size=57
operands init "dialog=dbox, size=100:120, h=40, v=80,":
                 "defpushbutton=102:OK:20:10, h=10, v=30, edit=101::30"
msg      list
msgname  char 8
msgtype  char 4
msgitem  num 5
msgdesc  num 40
         listend
contents char 30
         prepare window, "window=window01, size=100:100, pos=10:10"
         prepare dialog, operands
         link dialog, msgqueue
         show dialog, window
```

```
          loop
                    wait msgqueue
                    get msgqueue; msg
                    if (msgname = "dbox " and msgitem = 102)
                              query dialog, "status101"; contents
                              break
                    endif
          repeat
```

The list variable called **msg** consists of four variables: **msgname**, **msgtype**, **msgitem**, and **msgdesc**. These variables constitute the four parts of a message sent by a window.

The queue statement contains the **size** operand which defines the size (in characters) of each queue entry in **msgqueue**. In this example, the size of each message is 57 (eight characters for the resource name, four characters for the type, five characters for the item number, and 40 characters for the description).

The character variable contents is used to receive the contents of the edit field control.

At the start of the loop, the program is suspended until **msgqueue** is non-empty. When a message is available, the get statement retrieves it from the queue and places it in the list of variables specified by **msg**.

By default, no messages are sent when a user types characters into the edit field control.

If the user presses the **OK** push button, a message is placed in **msgqueue**. After the get statement is executed, the value of **msgitem** will be 102 because the push button was defined in the prepare statement with the item number 102.

If the value of **msgitem** is equal to 102 (the push button was pressed), execution continues with the query statement.

The query statement queries the status of the edit field control in the **dialog** resource. The result of the query is moved to **contents**.

The result of a status query for an edit field control is the text value of the edit field. Thus, the query statement moves the text from the edit field into the **contents** variable.

The break statement causes execution to continue with the statement that follows the repeat statement.

Images

An image variable defines a logical canvas for the draw statement to operate on.

The show statement displays an image in a window. The hide statement removes the image from display in the window.

Here is a simple example:

```
screen   device
imagevar image h=600, v=400, colorbits=8
         prepare screen, "window=window01, size=600:400, pos=10:10"
         show imagevar, screen, 21, 41
         draw imagevar; color=*blue, p=2:2, line=10:10
```

screen defines a window. **imagevar** defines an image with a size of 600 pixels by 400 pixels. Eight bits of color information (256 colors) are contained per pixel.

The show statement makes **imagevar** visible on **screen**. The upper left corner of the image is displayed at horizontal position 21 and vertical position 41 within **window01**.

In the draw statement, the **color** operand sets the current draw color to blue. The **p** operand sets the current draw position. The upper left corner of an image is position 1:1. The **line** operand draws a blue line from the current position to position 10:10. The new position is set to 10:10.

The next example is a paint program. When the user presses the left mouse button and moves the mouse, a line is drawn. The user can also press the right mouse button to change the draw color. Here is the program:

```
c_black  define 0
c_red    define 255
c_green  define 65280
c_yellow define 65535
c_blue   define 16711680
c_magenta define 16711935
c_cyan   define 16776960
c_white  define 16777215
win      device
image    image h=640, v=480
qmsg     list
name     char 8
func     char 4
item     num 5
horz     num 5
vert     num 5
         listend
msgq     queue entries=256, size=27
button   num 1
color    num " 1"
rgbcolor num 10[8], initial c_black, c_red, c_green:
                  c_yellow, c_blue, c_magenta, c_cyan, c_white
         prepare win, "window=window01, size=640:480, pos=10:10"
         link win, msgq
         draw image; color=*blue, erase
         show image, win
         loop
                  wait msgq
                  get msgq; qmsg
                  switch func
                           case "RBDN"
                                    add 1 to color
                                    if (color > 8)
                                             move 1 to color
                                    endif
                           case "LBDN"
                                    set button
                                    change win, "mouseon"
                                    draw image; color=rgbcolor[color]:
                                             p=horz:vert, dot
                           case "LBUP"
                                    change win, "mouseoff"
                                    clear button
                           case "POSN"
                                    if (button)
                                             draw image; line=horz:vert
                                    endif
                           case "CLOS"
                                    stop
                  endswitch
         repeat
```

The define statements assign RGB color values to the labels.

The variable **qmsg** defines the message that will be received when the user moves the mouse or clicks a button.

The message is made up of five variables: **name**, **func**, **item**, **horz**, and **vert**. These variables are the five parts of a message sent by the window.

name will always be **window01**. Some of the possible values of **func** are:

LBDN = left button down
LBUP = left button up
RBDN = right button down
RBUP = right button up
POSN = mouse movement report
CLOS = window closed from system menu

item always contains zeros. **horz** and **vert** contain the current mouse position.

The queue statement contains two operands. The **entries** operand specifies the number of messages allowed in **msgq**. The **size** operand defines the size (in characters) of each message. In this example, the size of each message is 27 (eight characters for the device name, four characters for the function, five characters that are always zero, five characters for the horizontal position, and five characters for the vertical position).

The prepare statement creates a window. The link statement links the window with the message queue. The show statement makes **image** visible in the win dow.

At the start of the loop, the program is suspended until **msgq** is non-empty. When a message is available, the get statement retrieves it and places it into **qmsg**.

If **func** is equal to **RBDN** (the right button was pressed), then one is added to the value of **color**. In other words, whenever the right mouse button is pressed, the draw color will change.

If **func** is equal to **LBDN** (the left button was pressed), the button variable will be set to 1. By default, only button action messages are sent to the queue. The change statement is used to alter this. The change statement that specifies **"mouseon"** causes all mouse movements to be reported by **POSN** messages. Finally, the draw statement is used to draw one pixel in the current draw color at the draw position specified by **horz:vert**.

If **func** is equal to **LBUP** (the left button was released), the **button** variable will be set to 0. The change statement that specifies **"mouseoff"** turns off **POSN** mouse position messages.

The **POSN** (the mouse was moved) message will only be reported if the left button is pressed down. If **func** is equal to **POSN**, the draw statement draws a line in the current draw color from the current draw position to the position specified by **horz:vert**.

If **func** is equal to **CLOS** (close action), the program will stop.

Compiling and Running Programs

DB/C DX programs are compiled, debugged and executed from the operating system command line and from batch files and shell scripts. This chapter describes how to do this.

Compiling Programs

The executable program that compiles DB/C DX source programs is **dbcmp**. This program creates **.dbc** object files from source programs.

Unless modified by the -8 or -9 command line option, the equate label named **DBC_RELEASE** is predefined with the value 101 and the equate label name **DBC_DX** is predefined with the value 1.

The format of the dbcmp command line is:

`dbcmp` *file1* [*file2*] [**`-c`**] [**`-d`**[**`=`***n*]] [**`-e=`***name*] [**`-err=del`**] [**`-f`**] [**`-h=`***string*] [**`-i`**] [**`-j`**[**`r`**]] [**`-k`**] [**`-v`**]
[**`-l=`***libraryname*] [**`-cfg=`***filename*] [**`-n=`***n*] [**`-o=`***filename*] [**`-p`**[**`=`***filename*]] [**`-r`**[**`=`***filename*]] [**`-s`**]
[**`-t=`***filename*] [**`-w=`***n*] [**`-x`**] [**`-z`**] [**`-afile=`***keyword*] [**`-file=`***keyword*] [**`-ifile=`***keyword*]
[**`-1`**] [**`-2`**] [**`-3`**] [**`-8`**] [**`-9`**]

file1	is the name of the source text file containing DB/C language statements. If the extension is not specified, **.txt** is assumed. The search path for the source file is controlled with the **dbcdx.file.source** runtime property.
file2	is the name of the output **.dbc** file. If *file2* is not specified, *file1* with an extension of **.dbc** is the default. If *file2* is specified without an extension, **.dbc** is assumed. The search and create path for the output file is controlled with the **dbcdx.file.dbc** runtime property.
`-c`	is the case insensitive option. Without this option, all source program labels are case sensitive. If this option is specified, case is ignored.
`-cfg=`*filename*	is the runtime properties file parameter. See the next chapter.
`-d`[**`=`***n*]	is the display compilation parameter. If specified, all lines from the source files are displayed on the output device. Statistics are displayed at the end of compilation. If this parameter is in the form of **-d**=*n*, then display of lines from the source files are suppressed until the program counter reaches line number *n*. When the program counter exceeds *n*, each line is displayed as with the normal **-d** parameter.
`-e=`*name*	is the define equate label parameter. The equate label specified by name is defined for this compilation and has a value of one.
-**`err=del`**	is the delete **.dbc** file on error option. If this option is specified, if any compilation error occurs, no **.dbc** file is created and any pre-existing **.dbc** is deleted. If this option is not specified, an invalid **.dbc** is created if compilation errors occur.
`-f`	is the formatted output parameter. This causes page formatting information, including header in formation, page numbers, and page breaks to be displayed.
`-h=`*string*	is the page header parameter. This parameter is only applicable if **-f** is specified.
`-i`	is the display information parameter. If this parameter is specified, the source file name will be displayed along with any errors or with the Compiled successfully message.
`-j`[**`r`**]	is the share open mode parameter. The **-j** parameter causes the input file to be opened in share read/write mode. The **-jr** parameter causes the input file to be opened in share read-only mode.
`-k`	is the check for odd comment parameter. If this parameter is specified, a warning message is generated in the following cases: if a right-side comment exists on a statement that allows an optional if and the if is not specified; and if a right side comment starts with a comma on any statement. Both of these situations may be caused by simple typing errors, but are technically correct programs.

`-l=`*libraryname* is the library parameter. The compiler checks the specified library for the existence of each include file. If the library does not exist or if the include file is not a member of the library, then the default search mechanism is used. The **-l** parameter can be specified one, two, or three times on the command line. The libraries are searched in the order they appear on the command line.

`-n=`*n* is the number of lines per page parameter. This parameter is only applicable if the **-f** parameter is also specified. If **-n** is not specified, the default number of lines per page is 56.

`-o=`*filename* is the options file parameter. *filename* is the name of a user-created file containing options. It is useful when the number of options is too large to fit on one command line.

`-p`[`=`*filename*] is the printer output parameter. If *filename* is specified, then all compiler output is written to that file. A **-p** parameter without *filename* is ignored.

`-r`[`=`*filename*] is the cross reference listing parameter. This parameter causes the compiler to create a cross reference listing that is displayed at the end of compilation. It contains a table with the name, definition, type, and usage of all variables. If *filename* is specified, then it is the name of the DB/C DX sort utility that is used in creating the cross reference table. If *filename* is not specified, then the default sort utility name is **sort.exe** in Windows and **sort** elsewhere.

`-s` is the statistics parameter. If specified, compilation statistics are displayed at the end of compilation.

`-t=`*filename* is the source line translate parameter. This parameter causes the compiler to translate each character of every source line. The translate table is contained in the file specified by *filename*. This file is a binary file that is 256 bytes long. The first byte is the replacement for a character with value 0, the second byte is the replacement for a character with value 1, etc.

`-v` is the verbose message display parameter. If this parameter is specified, the file includes and a running line count are displayed in addition to the information displayed by **-i**.

`-w=`*n* is the page width parameter. This parameter is only applicable if **-f** is specified. If **-w** is not specified, the default page width is 132.

`-x` is the external label parameter. If this parameter is specified, an external label definition may occur after the first use of that label. The **-x** parameter causes programs to execute somewhat more slowly than when **-x** is not specified.

`-z` is the debug parameter. This parameter causes a debug map file to be created in the same directory as the **.dbc** file. The name of this file is *file1***.dbg**. This file must exist for the debug parameter on the runtime command line (**-z**) to work.

`-afile=`*keyword* is the associative index file type parameter. This parameter overrides the default file type for an afile definition. If the **-afile** parameter is not specified, the standard DB/C file type is the default. *keyword* specifies which file type is the new default type. Valid keywords are: **text**, **data**, **native**, **crlf**, or **binary**.

`-file=`*keyword* is the file type parameter. This parameter overrides the default file type for a file definition. If the**-file** parameter is not specified, the standard DB/C file type is the default. *keyword* specifies which file type is the new default type. Valid keywords are: **text**, **data**, **native**, **crlf**, or **binary**.

`-ifile=`*keyword* is the indexed file type parameter. This parameter overrides the default file type for an ifile definition. If the **-ifile** parameter is not specified, the standard DB/C file type is the default. *keyword* specifies which file type is the new default type. Valid keywords are: **text**, **data**, **native**, **crlf**, or **binary**.

The -1, -2, and -3 parameters control how the include file names are used by **dbcmp**. They are mutually exclusive.

`-1` is the truncate end parameter. This parameter causes the file name in each include statement to be truncated at the first slash. All characters in the file name following the slash (as well as the slash itself) are not used. For example: **inc abc/def** is converted to **inc abc**.

`-2`	is the period replacement parameter. This parameter causes the first slash in the file name in each include statement to be changed to a period. For example: **inc abc/def** is converted to **inc abc.def**.
`-3`	is the truncate beginning parameter. This parameter causes the file name in each include statement to be truncated at the first period that follows the first slash. All characters in the file name before the period (as well as the period itself) are not used. For example: **inc ab/de.ghi** is converted to **inc ghi**.
`-8`	is the DB/C release 8 compiler compatibility parameter. This parameter causes the **DBC_RELEASE** and **DBC_DX** equate label to be undefined, the syntax of the trap statement with the queue operand to work as in DB/C release 8, and xif statement processing to work as in DB/C release 8.
`-9`	is the DB/C release 9 compiler compatibility parameter. This parameter causes the **DBC_RELEASE** equate label to be defined with the value 9, the **DBC_DX** equate label to be undefined, and xif statement processing to work as in DB/C release 9.

Running Programs Without Smart Client

Compiled programs (**.dbc** files) are run by the program execution facility, commonly called the runtime. The name of the runtime is **dbc.exe** in Windows and **dbc** elsewhere. There is an additional runtime for Windows named **dbcc.exe** that does not allow GUI applications and can be run as a service (see Chapter 10).

The format of the runtime command line is:

`dbc` [*file1*] [**`-cfg=`***filename*] [**`-l=`***libraryname*] [**`-o=`***filename*] [**`-z`**] [*parameters*]

file1	is the name of a **.dbc** object file to be run. If *file1* is not specified, **answer.dbc** is assumed. If *file1* is specified without an extension, **.dbc** is assumed.
`-cfg=`*filename*	is the runtime properties file parameter. See the next chapter.
`-l=`*libraryname*	is the library parameter. When this parameter is specified, the *libraryname* file is searched first for the name of the program object file. If the library does not exist or the file is not a member of the library, then the default search mechanism is used. **-l** may be specified up to 16 times. If specified more than once, the order of search is from left to right on the command line.
`-o=`*filename*	is the options file parameter. *filename* is the name of a user-created file containing command line options. It is useful when the number of options is too large to fit on one command line.
`-z`	is the debug parameter. It makes the source code debugger available.
`-zs=`*portnumber*	This is a special debug parameter that is used only for remote debugging.
parameters	is one or more additional user-defined parameters used by DB/C programs. One or more parameters may be specified. A parameter must start with an alphanumeric character.

There are a number of runtime properties that control the behavior of the runtime. The runtime properties are contained in the DB/C DX runtime properties file. By default, the name of this file is **dbcdx.cfg** and it is located in the current directory. The name of this file may be overridden with the **-cfg** command line parameter or with the **DBC_CFG** environment variable. See the next chapter.

Running Programs With Smart Client

Compiled programs (**.dbc** files) can be run in a client/server mode called Smart Client. This feature allows for remote execution of the user interface statements of DB/C programs. That is, for character mode programs (those that use keyin and display statements), the client program of Smart Client (for example **dbcscc.exe**) is connected to a process on a server that is executing the DB/C DX runtime (for example **dbc** in Linux) via TCP/IP. For GUI programs, the Windows GUI Smart Client program (**dbcsc.exe**) and the Java Smart Client program both implement the GUI user interface capabilities of the DB/C program that is running on the DB/C DX runtime on the server as a process on Linux or on Windows server.

There is a 'keep alive' feature that is implemented between Smart Client server and client software. If the TCP/IP connection between the client and server is broken or is unavailable for more than a minute, the runtime will shut itself down.

Before the client software of Smart Client is started, the Smart Client server dispatcher must be running on the server machine. The name of this program is **dbcd.exe** for Windows and **dbcd** elsewhere.

Starting the Smart Client Server Dispatcher

The syntax to start the Smart Client server dispatcher from the command line is:

`dbcd` [**`-dx=`***executable*] [**`-port=`***nnn*] [**`-sport`**[=*nnn*]] [**`-encrypt=`***option*] [**`-cd=`***directory*]

`-dx=`*executable* specifies an alternate name for the DB/C DX runtime that will be started when each client connects. The default runtime for Windows is **dbcc.exe** and **dbc** elsewhere. The value of this operand is the name of an executable file.

`-port=`*nnn* specifies the TCP/IP port number for the initial connection with the Smart Client dispatcher (**dbcd**). The default value is 9735.

`-sport`[=*nnn*] specifies the port number that the Smart Client uses for the secondary connection to the dbc task started by the dispatcher. If the port number (*nnn*) is not specified, the dispatcher will dynamically allocate one. See the **-localport** option of **dbcsc** for more information. If **-sport** is specified, the Smart Client acts as the TCP/IP client for both initial and secondary connections. If **-sport** is not specified, the Smart Client acts as the TCP/IP server for the secondary connection.

`-encrypt=`*option* specifies the policy that the dispatcher will use for encrypted connections. *option* may be **on**, **off** or **only**. If **only** is specified, the dispatcher will not allow connections from clients that have encryption disabled. If **off** is specified, the dispatcher will expect clients to connect without encryption, and not allow encrypted connections. If **on** is specified, the dispatcher will allow both encrypted and non-encrypted connections; this is the default. See the section titled Considerations for Running Smart Client with the -encrypt Option.

`-cd=`*directory* is used to specify the directory that the dispatcher will change to when it starts.

Installing DBCD as a Windows Service

The Windows server dispatcher (**dbcd.exe**) may be installed as a service in Windows NT, Windows 2000, and newer versions of Windows. The following are additional command line options for installation and removal of the Smart Client server dispatcher as a Windows service:

`-display=`*name* is used with the -service parameter and defaults to DB/C DX Smart Client Service.

`-install` is the install parameter. This parameter causes **dbcd** to be installed as a service. The default is to install the service in a stopped state. See **-start**.

`-password=`*password* is the login password parameter and is optionally used with the **-user** parameter.

`-service=`*name* is useful if you wish to run multiple **dbcd** dispatchers as services concurrently on the same system. The default value is **DbcdService** and must be different for the additional instances of **dbcd** services. You should also change the service display name using the **-display** option.

`-start` is the start parameter which starts the service. This can also be accomplished from the Services Control Panel or by restarting the computer. This option may be used in conjunction with the **-install** parameter or by itself to start a stopped service.

`-stop` is the stop parameter which stops the service. This can also be accomplished from the Services Control Panel.

`-uninstall` causes the service to be uninstalled. It performs an implicit -stop if necessary.

`-user=`*logonuser* is used to assign a user to the service. *logonuser* should be in the format of *domain-name*\*user-name*. If this parameter is not specified, then the **LocalSystem** account will be used. This can also be accomplished with the Services Control Panel prior to starting the sevice. A user name may need to be defined drive mappings that are not available with the LocalSystem account. If the user name requires a password, the **-password** parameter should also be specified.

`-verbose` causes various success or failure messages to be displayed.

The Services Control Panel can be used to start, stop and check the running status of the Smart Client dispatcher service. If an error occurs which causes **dbcd.exe** to stop running, the error will usually be logged in the Windows Application Log which can be viewed with the Event Viewer.

Running the Smart Client Dispatcher as a Linux Background Process

When executing the Smart Client server dispatcher as a background process in Linux, specify the **-y** option to cause **dbcd** to continue to run even after the terminal connection used to start the dispatcher is closed. This causes display of information normally sent to **stdout** to be redirected to **/dev/null**. Here is what the command line might look like:

```
dbcd -y &
```

Starting the Smart Client Application

After the server dispatcher is running, clients may connect by starting one of the Smart Client client programs. There are several client programs. **dbcscc.exe** is the Windows console mode version of the client program. **dbcsc.exe** is the Windows GUI mode version of the client program. **dbcsc** is the Linux character mode version of the client program. The Java class **com.dbcswc.sc.Client** in **dbcsc.jar** is the Java Smart Client program, and which is the start up program used instead of **dbcsc**.

The syntax to start the client software from the command line is:

`dbcsc` *computer* [**`-hostport=`***nnn*] [**`-localport=`***nnn*] [**`-t=`***tdbfilename*] [**`-encrypt=`***option*] [**`-user=`***username*] [**`-curdir=`***directory*] [**`-a=`***nnn*] [**`-pl`**] [**`-lp`**] [**`-fontsize=`***nn*] [*parameters*]

computer is the IP address or DNS name of the computer running the server dispatcher.

`-hostport=`*nnn* is the TCP/IP port number on which the client connects to the server dispatcher. It must correspond with the **-port** operand on the server dispatcher command line.

`-localport=`*nnn* is the port number that the runtime on the server uses for the secondary connection to the Smart Client. If **-localport** is not specified, a default value is dynamically assigned as the port number, typically greater than 10000. If **-localport** is specified with the value 0 and the server dispatcher was started with the **-sport** option, then the secondary connection is reversed, meaning that instead of the client waiting for a secondary connection from the runtime, the client will make the secondary connection to the runtime. The value 0 is very useful for when the client is behind a firewall or NAT boundary (that is, the client doesn't have an IP address accessible from the server).

`-t=`*tdbfilename* is a Linux option used to specify the terminal definition file for the Smart Client session.

`-encrypt=`*option* specifies if encryption will be used for the TCP/IP connection ot the runtime. If the value of *option* is **off**, encryption will not be used. If the value of *option* is **on**, encryption will be used. The default is **on**. See the section titled Considerations for Running Smart Client with the -encrypt Option.

`-user=`*username* is a Linux option that specifies the effective user that the runtime process will run as on the server. To make this option work, **dbcd** must be owned by root, and must have the set user ID on execution bit set. Use the following commands to accomplish this:

```
chown root:root dbcd
chmod u+s dbcd
```

`-curdir=`*directory* is an option that specifies the current directory on the server for the runtime.

`-a=`*nnn* specifies the amount of memory allocated for the internal work area. *nnn* is kilobytes.

`-pl` specifies that the server uses the Latin1 character set, and that these characters should be mapped to PC BIOS characters when received by the client. This option would typically be used when the server side is Linux and Smart Client is running under Windows, and international characters are involved in keyin and display statements.

`-lp` specifies that opposite configuration compared with **-pl**. That is, the server side is Windows using the PC BIOS characters, and a Linux Smart Client is being used.

parameters is one or more additional user-defined command line options used by DB/C programs. These options are passed to the runtime at start of execution. The first of these options corresponds with the startup **.dbc** program. If none is specified, the default is **answer.dbc**.

The command line options apply to the Java and .NET Smart Client programs where they are passed as parameters to the **main** method in the start up class.

All command line options specified above are absorbed by each of the different Smart Client programs, including those ignored by a particular Smart Client. All other command line options (*parameters*) are passed through to the DB/C DX runtime command line that is started on the server.

For the Java Smart Client, the **-encrypt** option will only work if the Java Secure Sockets Extension (JSSE) support libraries are available to the JVM.

If there is a firewall between the client and the server, TCP/IP ports must be available for connections. At startup, the Smart Client client program connects to the server dispatcher on the port specified by **-hostport** on the client and **-port** on the server (the default is 9735). The dispatcher then starts a DB/C DX runtime which connects back to the client on the port specified by **-localport** or on a dynamically assigned port number if **-localport** isn't specified.

Considerations for Running Smart Client with the -encrypt Option

Encryption for Smart Client uses industry standard SSL key managment. A key file containing a certificate and a private key for the certificate need to be generated and stored in a key file.

Here is an example of the commands to make the key file on Linux (substitute your own location and company name in the string following **-subj**):

```
openssl genrsa -out selfsigned.key 4096

openssl req -new -key selfsigned.key -out selfsigned.csr  -sha256 \
    -subj "/C=US/ST=Wyoming/L=SelfSigned/O=Portable Software/OU=Org/CN=localhost"

openssl x509 -req -days 9000 -in selfsigned.csr -signkey selfsigned.key -out selfsigned.crt

cat selfsigned.crt selfsigned.key > dbcserver.crt
```

Here is an example of the Power Shell commands to make the key file on Windows (substitute your own location and company name in the string following **-subj**):

```
openssl genrsa -out selfsigned.key 2048     // 4096 is an option

openssl req -new -key selfsigned.key -out selfsigned.csr -sha256
        -subj "/C=US/ST=Wyoming/L=SelfSigned/O=PTSW/OU=Org/CN=localhost"

openssl x509 -req -days 9000 -in selfsigned.csr -signkey selfsigned.key -out selfsigned.crt

Get-Content .\selfsigned.crt, .\selfsigned.key | dbcserver.crt
```

The key file (**dbcserver.crt** in the examples) can have any name, but **dbcserver.crt** is the default name that the the DB/C DX Smart Client server looks for. By default, that file should be located in the current directory that is effective with the server starts. The file name and location can be modified with the **dbcdx.smartserver.certificatefilename** runtime property.

For the Java Smart Client, the **-encrypt** option will only work if the Java Secure Sockets Extension (JSSE) support libraries are available to the JVM.

Compiling, Running and Debugging Programs using Eclipse

DB/C DX programs may be developed using the Eclipse with the DB/C Development Toolkit (DDT) feature for Eclipse. Eclipse with DDT provides these capabilities:

- editing of DB/C source programs with language-specific features like syntax coloring and hover help
- automatic compilation and source code editor integration of compilation error messages
- runtime execution integration
- source level debugging of DB/C programs
- integration with source code control systems

Eclipse is an open source Integrated Development Environment (IDE) framework that works on Windows, MacOS, and LINUX.

The first step to using DDT and Eclipse is to install Eclipse. Eclipse may be downloaded from **www.eclipse.org**. Install Eclipse according to the instructions for your operating system.

DDT is distributed with DB/C DX in the **ddt.zip** file. Move that file to a directory of your choice. DDT can then be installed using the standard Install Features menu items of Eclipse by specifying the **ddt.zip** file as an Update Site and then completing the installation normally.

DB/C DX projects may be created using the DX Perspective. Programs and includes are created as files with specific file name extensions. Compilation occurs automatically on file save of programs and when the project is built using the Build menu option. Eclipse debugging features like breakpoints and code stepping are the same as for other languages.

Debugging Programs in Character Mode

To use the character mode source program debugger, programs need to be compiled with the **-z** option on the compiler command line. When this option is used, a **.dbg** file is created in addition to the **.dbc** file.

When the **-z** command line option is specified on the runtime command line, the character mode source program debugger is activated to control program execution.

The debugger window is divided into five subwindows stacked from top to bottom. The top subwindow is the command entry area. When the debugger window is active, the cursor is always blinking in the command entry area waiting for user keyboard input. The second subwindow is a variable height subwindow that contains the source program. The next subwindow is one line that contains execution status information. The next subwindow is the variables subwindow. This variable height subwindow contains view variables (explained be low). The bottom subwindow is the results subwindow. This variable height subwindow displays the result of each debugger command.

One of the three variable height subwindows (source, variables and results) is active. The active subwindow has its title displayed in reverse color (black on white). Several cursor keys (home, end, page up, page down, up, down) control the display in the active subwindow. The Window command (press the W key) toggles between the three titled subwindows. The Help command (press H or ?) displays a list of all commands in the results subwindow.

Here is a description of each command:

`a`	Alter flags - this allows each of the flags to be set, cleared or remain unchanged.
`b`	Permanent breakpoint - this sets a permanent breakpoint at the line highlighted in the source window. The breakpoint will remain active until the module is unloaded. Permanent breakpoints are denoted with an asterisk to the left of the line number. If the highlighted line is a debug statement, pressing **b** once will disable the debug statement. pressing **b** again will enable it.
`c`	Clear breakpoints - this causes each of the permanent, temporary and value breakpoints to be cleared.
`d`	Display variable - this displays the current value of a variable in the results window.
`e`	Extended command - this prompts for one of the extended commands (see below).

`f`	Change source file - this prompts for a source file name to be displayed in the source window.
`g`	Go - resume execution.
`h` and ?	Help - displays a listing of the commands in the results window.
`i`	Set tab indent - prompt for the number of spaces that leading tab characters are replace with in the source window.
`j`	One up the return stack - this causes the debug pointer in the return stack to be moved so that the return stack seems one entry smaller. The line where that return statement would return to is displayed as the current line in the source window.
`k`	One down the return stack - this causes the debug pointer in the return stack to be moved so that the return stack seems one entry larger. The line where that return statement would return to is displayed as the current line in the source window. If at the end of the return stack, then the next execution line is displayed as the current line in the source window.
`l`	Find program label - this prompts for a program label to search for in the source file displayed in the source window.
`m`	Modify variable - this prompts for a variable to be modified and for its new value. The variable to be modified must be in the current module.
`n`	Change module - this prompts for a new module to become the current module.
`o`	Display output screen - display the current output screen waiting for any character to return to the debugger. This is useful for the character mode version of DB/C.
`p`	Break on value - this prompts for a variable name and break condition for the variable. When the variable satisfies the condition a break occurs and the break on value is cleared.
`q`	Shutdown - this immediately terminates the debugger and the program being debugged.
`r`	Remove view variable - this removes a variable from the variables window.
`s`	Search for string - this searches the current source file for a string of characters. This string may be in any line including comment lines.
`t`	Temporary breakpoint - set a temporary breakpoint at the line highlighted in the source window. Temporary breakpoints are automatically cleared the first time program execution reaches and is paused by a temporary breakpoint. Temporary breakpoints are denoted with a plus sign to the left of the line number.
`u`	Step out of call - resume execution and then pause on the first statement executed after the current return stack scope is removed from the stack. This may happen via return statement or by various other methods.
`v`	Add view variable - this prompts for a variable name and then adds a continuously updated display of that variable in the variables window.
`w`	Change window - toggle the active window between the source, variables and results windows.
`x`	Single step - the next line of the source program will execute.
`y`	Go to line number - cause the highlighted line in the current source file to be changed to the line that is specified by the line number that is prompted for.
`z`	Step over call - one line of the source program will execute unless the line is a call statement, in which case execution continues until it is paused at the line immediately following the call statement.
`-`	Decrease window size - shrink the height of the active window of the source, variables or results window.
`+`	Increase window size - enlarge the height of the active window of the source, variables or results window.

`#`	Toggle line numbers - toggle the line numbers in the source window either off or on. Line numbers are initially on.
`%`	Toggle case sensitive - toggle case sensitivity for all variable names, labels and search strings. Case sensitivity is initially off.
`/`	Search again - this causes the last search command to be repeated.

These extended commands are available at the **EXTCMD:** prompt:

`a`	Set a breakpoint that will occur only when the value of a variable changes. This prompts for a variable name. The current value of variable is captured, and a breakpoint will happen when the value of the variable changes. The changed value of the variable will then be captured so that continuing execution will cause a break to occur only when the value changes again.
`c`	Clear global variables - all global character and numeric variables are cleared.
`g`	Save global variables - all global character and numeric variables are saved in the **dbgglbl.cfg** file. The default file name can be altered with the **dbcdx.debug.globa**l runtime property.
`l`	Load global variables - global character and numeric variables are restored to the values saved in the **dbgglbl.cfg** file.
`m`	Toggle more option - toggle the single page pause mode of the results window. This mode is initially off.
`p`	Save preference - save the debugger preference settings in the **dbgpref.cfg** file. The default file name can be altered with the **dbcdx.debug.preference** runtime property.
`r`	Toggle result window - the results window will toggle between display and invisible mode. It is initially displayed.
`v`	Toggle variables window - the variables window will toggle between display and invisible mode. It is initially displayed.

Runtime Properties

The DB/C DX runtime properties file contains configuration and runtime information for execution of DB/C DX programs. The properties file is a native text file that contains key word value pairs in each line of the file. Properties files use the backslash character (\) as an escape character. Thus when you need to include one backslash in a value, specify two consecutive backslashes.

This search for the runtime properties file is done in the following order:

1. **-cfg=***filename* option specified on the command line.
2. **DBC_CFG=***filename* defined in the operating system environment.
3. the file named **dbcdx.cfg** located in the current directory.

These elements are used in the following descriptions:

program is the name of a .dbc file created by the DB/C DX compiler.

string is a string of characters and digits. It may contain blanks.

number is a string of one or more digits.

directory-or-fsname is either a directory name or is the DB/C FS server name as specified in the **dbcdx.file.server.fsname.** properties. If it is a directory name, it may contain blanks and other special characters.

directory-or-fsname [`;` *directory* …] is either a single DB/C FS server name or one or more directories separated by a semicolon. Combining a DB/C FS server name with one or more directories is not supported.

translate-spec is either *nnn*`:`*nnn* or *nnn*`-`*nnn*`:`*nnn* where *nnn* is the decimal value of a character. The value before the colon is the translate from character or the translate from character range, and the character after the colon is the translate to character or the first character in the translate to range.

The following keyword-value pairs of properties may be specified. Unless otherwise noted, each keyword may only be specified once in the properties file.

`dbcdx.beep = old`

When this property is specified on a Windows machine, beep in **dbc.exe** works as it did in DB/C DX Release 11.0, and does not use the Windows default alert sound.

`dbcdx.bitop = old`

When this property is specified, the and, not, or, and xor statements always clear the high order bit as in DB/C release 8.0.

`dbcdx.compare = old`

When this property is specified, the compare statement sets the over flag if there would be overflow as in DB/C release 7.0.

`dbcdx.console = file`

This property defines the file name of the console file if the console statement is used. If this property is not defined, then the console statement is ignored.

`dbcdx.cxcompat = old`

This property supresses the E 566 error.

`dbcdx.errordisplay = off`

This property specifies that an error message will not be displayed when there is an untrapped error.

`dbcdx.keytag = old`

If this runtime property is specified, the keytag property of the reformat and sort utilities will be the same size as in DB/C release 8.0, which were 8 and 9 bytes respectively.

`dbcdx.kydspipe = on`

This property causes the **`*`p=***n*`:`*n*, **`*`resetsw**, **`*`hoff** and **`*`alloff** control codes to be ignored. In addition, the cursor is never turned off. The DB/C DX runtime implements these control codes by directing a sequence of escape characters to the screen. When the output is being redirected to a Linux pipe, these escape sequences can cause problems. The **dbcdx.kydspipe** property is useful in this situation.

`dbcdx.memalloc =` *nnn*
This property specifies the number of kilobytes that will be allocated for program area, data area, and other buffers. The default is 2048. This value should be increased if an out of memory error occurs.

`dbcdx.precision = old`
When this property is specified, the divide statement and divide expressions will give results with the same precision as in DB/C release 8.0.

`dbcdx.preload =` *program* [`;` *program* …]
This property specifies the modules to be preloaded. Each program is a **.dbc** file that was created by the compiler. A single, unnamed instance of each program is created before any other program execution begins. The **.dbc** extension does not need to be specified. A program specified by this property may not be used as a target of a loadmod statement.

`dbcdx.rounding = old`
When this property is specified, rounding works as in DB/C release 7.0.

`dbcdx.search = old`
When this property is specified, the search statement will compare two numeric values by comparing the logical strings as in DB/C release 8.0.

`dbcdx.start =` *program*
This property specifies the default startup program. If not specified, **answer.dbc** is the default startup program. Specification of this property changes the behavior of **dbcdx.stop**.

`dbcdx.stop =` *program*
This property specifies the default stop program. If not specified, the stop statement or an untrapped error causes execution to terminate. If specified, a stop statement or an untrapped error occurring while executing a program other than then the program(s) specified by **dbcdx.start** and **dbcdx.stop** will cause the program specified as the stop program to be chained to. If the program executing the stop or untrapped error is the same as the program specified by **dbcdx.stop** and **dbcdx.start** is specified, then the program specified by **dbcdx.start** is chained to. Otherwise, the stop statement or an untrapped error causes execution to terminate.

`dbcdx.trap = old`
If this runtime property is specified, then a trap automatically assumes the trap properties nocase and prior as in DB/C release 6.0.

`dbcdx.`*utility* `=`*filename*
utility is one of: **aimdex**, **build**, **copy**, **create**, **encode**, **exist**, **index**, **reformat**, or **sort**. This property overrides the runtime operation of performing the operation internally or submitting the file manipulation command to the DB/C FS server if **dbcdx.file.primary.server** is specified. This property causes the runtime instruction of the same name to execute the program name defined by *filename* in a manner similar to rollout without any display.

`dbcdx.client.memalloc =` *nnn*
This property specifies the number of kilobytes that will be allocated for images, buffers and other information in the Smart Client client. This value is overridden by the **-a** command line option on the Smart Client command line. This value should be increased if an out of memory error occurs.

`dbcdx.smartserver.certificatefilename =` *filename*
This property specifies the file name of the key certificate file to be used for encrypted communication with Smart Client clients. If not specified, the default file is **dbcserver.crt** in the current directory.

`dbcdx.clock.day =` *21-char-string*
Each three characters specifies the day returned by the clock statement. The default value is **"SunMonTueWedThuFriSat"**.

`dbcdx.clock.env =` *string*
This property specifies the string value returned by the clock env statement. The default value is the operating systems environment.

`dbcdx.clock.month =` *36-char-string*
Each three characters specifies the month returned by the clock statement. The default value is **"JanFebMarAprMayJunJulAugSepOctNovDec"**.

`dbcdx.clock.port =` *string*
This property specifies the string value returned by the clock port statement.

`dbcdx.comm.tcp.serverkeepalive = on`
This property causes the tcp sockets that are created by TCP/IP **comfile** statements to be created with the "SO_KEEPALIVE" option.

`dbcdx.debug.global =` *filename*
This runtime property defines the character mode debugger's default file name for saving the value of global variables. If not defined the default is **dbgglbl.cfg** in the path defined by **dbcdx.file.dbc**. This property is only applicable to the character mode debugger.

`dbcdx.debug.preference =` *filename*
This runtime property defines the character mode debugger's default file name for saving the debugger preference. If not defined the default is **dbgpref.cfg** in the path defined by **dbcdx.file.dbc**. This property is only applicable to the character mode debugger.

`dbcdx.display.autoroll = off`
This property specifies that, except for the **`*`r** control code, the display statement does not cause the current window to scroll up.

`dbcdx.display.bgcolor =` *color*
color is one of **`*`red**, **`*`white**, **`*`black**, **`*`blue**, **`*`green**, **`*`yellow**, **`*`cyan** or **`*`magenta**. This property specifies the default background color. If *color* is not specified, the background color defaults to black.

`dbcdx.display.color =` *color*
color is one of **`*`red**, **`*`white**, **`*`black**, **`*`blue**, **`*`green**, **`*`yellow**, **`*`cyan** or **`*`magenta**.. This property specifies the default foreground color. If *color* is not specified, the foreground color defaults to white.

`dbcdx.display.columns =` *number*
This property specifies the number of display columns.

`dbcdx.display.console.closebutton = off`
This property specifies that the console window will not have a close button.

`dbcdx.display.device =` *devicename*
This property specifies the output device or file associated with display operations.

`dbcdx.display.font =` *fontname*
This property specifies the font for the graphical keyin and display window. The default is a monospaced font.

`dbcdx.display.hoff = old`
If this runtime property is specified, an **`*`alloff** (*__hoff__) control code in a display or keyin statement causes the foreground and background colors to be reset to the default in the sammer manner as in DB/C release 7.0.

`dbcdx.display.lines =` *number*
This property specifies the number of display lines.

`dbcdx.display.resetsw = old`

When this property is specified, the display control code ***resetsw** will function in the same manner as in DB/C Release 7.0 and will not alter the cursor position.

`dbcdx.display.termdef =` *filename*

This property specifies the termdef binary file that defines the terminal characteristics. In Linux, if this property is not specified, the Linux terminfo capability is used.

`dbcdx.file.aiminsert = old`

This property causes the AIM write and insert operation to not invalidate the readkg or readkgp index position as specified by the PL/B ANSI standard. A readkg or readkgp occurring with an invalid index position will cause an I 714 error to occur. Setting this property will cause DB/C DX to be similar to DB/C 9 and prevent the aim write or insert from in validating the index position.

`dbcdx.file.casemap =` *translate-spec* [`;` *translate-spec* …]

This property specifies the file case translation table used to modify the default behavior for case insensitive aim reads. This property may be needed to support international character set as DB/C DX by default only converts **a-z** to **A-Z** for aim record comparisons. This would be equivalent to having a translate value of 97-122 : 65.

`dbcdx.file.collate =` *translate-spec* [`;` *translate-spec* …]

This property specifies the collating sequence table used to modify the default behavior of the sort utility and index keys. This property may be needed to support international character sets as DB/C DX by default sorts using the ASCII value of the character. This would be equivalent to having a translate value of 0-255 : 0.

`dbcdx.file.compat = dos`
`dbcdx.file.compat = rms`
`dbcdx.file.compat = rmsx`

This property specifies the Datapoint DATABUS file statement compatibility. This property causes the file name in open, prepare and utility statements to be altered before the operation occurs. If **dos** is specified, the first slash (/) in the file name is translated to a period. If **rms** is specified, the first slash (/) in the file name is translated to a period. If the extension is **TEXT**, it is translated to **TXT**. If the extension is **ISAM**, it is translated to **ISI**. In addition, if the name portion of the file name (left of the slash) is longer than eight characters, it is truncated to eight characters. If **rmsx** is specified, the same actions occur as for **rms**, except name portion truncation does not occur.

`dbcdx.file.dbc =` *fsname*
`dbcdx.file.dbc =` *directory* [`;` *directory* …]

This property specifies the name of the DB/C FS server or the local directories to search when opening a program object (**.dbc**) file. If the value is a directory or a list of directories an attempt is made to locate that file in the directory or directories in the order specified. If the value is a DB/C FS server name, the database server will attempt to locate the file in the directory or directories that is controlled by the database configuration information.

`dbcdx.file.editcfg =` *directory*

The source code editor utility (edit) stores configuration information in a file named **edit.cfg** in the current directory. If this runtime property is specified, the directory location of this file is designated by directory.

`dbcdx.file.excloffset =` *nnn*

This property specifies the file offset that will be used to implement the exclusive open operation. The default value is operating system dependent, and should usually not be changed.

`dbcdx.file.exclusive = off`

This property applies to the Linux runtime and utilities. It causes file open to ignore any file locks.

`dbcdx.file.extcase = upper`

This property specifies how default extensions are appended to file names that do not contain an extension. If this property is specified, an uppercase extension is appended to the file name. If this property is not specified, a lowercase extension is appended to the file name.

`dbcdx.file.fileoffset =` *nnn*
This property specifies the file offset that will be used to implement the filepi operation. The default value is operating system dependent, and should usually not be changed.

`dbcdx.file.filepi = noerr`
If this runtime property is specified, an implicit **filepi 0** occurs before each filepi statement. This will inhibit E 504 errors from occurring when the scope of two filepi statements overlap.

`dbcdx.file.fonts =` *directory* [**`;`** *directory …*]
If property specifies the name of the local directories to search for font files used to create PDF and PS print files. It is only used in Linux environments.

`dbcdx.file.ichrs = on`
This property allows DB/C-type files to contain characters in the range 128 through 248. Digit compression is disabled, but space compression still works. If this property is not specified, then values 128 through 248 are considered to be digit compression characters.

`dbcdx.file.image =` *fsname*
`dbcdx.file.image =` *directory* [**`;`** *directory …*]
This property specifies the name of the DB/C FS server or local directories to search when opening an image file. If the value is a directory or a list of directories an attempt is made to locate that file in the directory or directories in the order specified. If the value is a DB/C FS server name, the database server will attempt to locate the file in the directory or directories that is controlled by the database configuration information.

`dbcdx.file.keytrunc = on`
If this runtime property is specified, an index key with a length greater than the specified key length is truncated with no regards to the characters truncated as in DB/C release 8.0.

`dbcdx.file.minusoverpunch = ascii`
If this runtime property is specified, the characters that are written to disk when the **`*mp`** control code is specified are ASCII overpunch characters.

`dbcdx.file.lock = sem`
If this runtime property is specified, the filepi statement and record locking uses Linux semaphores instead of Linux file locks. This property is ignored in Windows.

`dbcdx.file.nameblanks = nochop`
`dbcdx.file.nameblanks = squeeze`
This property changes the file name processing behavior in the open and prepare statements. The default file name processing behavior is to chop trailing blanks from the file name and keep existing blanks within the file name. Setting this property to nochop will cause no blank character conversions to occur. Setting this property to squeeze will cause all blanks to be removed from the file name.

`dbcdx.file.namecase = upper`
`dbcdx.file.namecase = lower`
If the value is a **`upper`**, then all file names are translated to upper case. If the value is **`lower`**, then all file names are translated to lower case.

`dbcdx.file.open =` *fsname*
`dbcdx.file.open =` *directory* [**`;`** *directory …*]
This property specifies the name of the DB/C FS server or local directories to search when opening a file. If the value is a directory or a list of directories an attempt is made to locate that file in the directory or directories in the order specified. If the value is a DB/C FS server name, the database server will attempt to locate the file in the directory or directories that is controlled by the database configuration information.

`dbcdx.file.openlimit =` *nnn*
nnn is the number of files that may be open concurrently at the operating system level. The default is dependent on the operating system configuration.

dbcdx.file.prefix. *fsname* **=** *fileprefix*

This property specifies that file names that begin with the directory *fileprefix* and do not have a volume specification as part of their name will be opened on the DB/C FS file server specified by *fsname*. The prefix is case sensitive for Linux, and case insensitive for Windows.

dbcdx.file.prep = *directory-or-fsname*

This property specifies the local directory or DB/C FS server where a file is created. If the value is a directory, then the file is created in that directory. If the value is a file server name, then the file is created in the directory that is controlled by the database configuration information.

dbcdx.file.primary.server = *fsname*

This property specifies the preferred DB/C FS server for all file operations, including file open, file prepare and execution of file utilities (e.g. aimdex). This property is ignored for file open and file prepare if the file name contains a volume specification or the property file contains **dbcdx.file.open** or **dbcdx.file.prep**.

dbcdx.file.prt = *fsname*

dbcdx.file.prt = *directory* [**;** *directory* …]

This property specifies the DB/C FS server or local directories to search when opening or creating a spool file with the splopen statement. If this property is not specified, then the splopen statement uses the current directory, not the open or prep property.

dbcdx.file.recoffset = *nnn*

This property specifies the file offset that will be used to implement record locking. The default value is operating system dependent, and should usually not be changed.

dbcdx.file.rolloutclose = on

This property specifies that rollout statement implicitly closes all shared files when the statement is executed. These implicit closes are apparent to the operating system, but are not apparent to the DB/C program, except to the extent that record locks are lost. This property is provided for compatibility with older versions of DB/C DX.

dbcdx.file.server. *fsname* **.database =** *database*

This property associates the database name to the user defined *fsname*. It is a required property for each **dbcdx.file.server.** *fsname* **.server** that is specified. database is the name of the data dictionary file (**.dbd**), which minimally must contain an access code appropriate for the values of the required **dbcdx.file.server.** *fsname* **.user** and **dbcdx.file.server.** *fsname* **.password**.

dbcdx.file.server. *fsname* **.encryption = on**

This property enables the encryption feature to the user defined *fsname*. It is an optional property for each **dbcdx.file.server.** *fsname* **.server** that is specified and is ignored if connecting to a DB/C FS 2 server. If not specified, then encryption will not be used.

dbcdx.file.server. *fsname* **.localport =** *nnn*

This property associates a local IP port number to the user defined *fsname*. It is an optional property for each **dbcdx.file.server.** *fsname* **.server** that is specified and is ignored if connecting to a FS 2 server. After the initial TCP/IP connection to the FS server, the local port number is used for the second TCP/IP connection which occurs in the direction from the FS server to the DB/C DX runtime. If this property is not specified, then a dynamic value will be used. If *nnn* is a positive value, it must not conflict with reserved values for the specific operating system. The value zero is a special case and causes the second TCP/IP connection to occur in the direction of DB/C DX runtime to FS server and sport must be defined in the FS server configuration file.

dbcdx.file.server. *fsname* **.password =** *password*

This property associates the *password* to the user defined *fsname*. It is an optional property for each **dbcdx.file.server.** *fsname* **.server** that is specified. If not specified, then a value of **PASSWORD** is used. password combined with the value defined by **dbcdx.file.server.** *fsname* **.user** needs to match a user name in the DB/C FS Server password configuration file.

dbcdx.file.server.*fsname***.server =** *computer*

This property associates the user defined *fsname* with the computer running the DB/C FS file server software. computer is the name of the computer (the DNS name) or is the IP address of the computer. Each specification of this property requires a matching **dbcdx.file.server.***fsname***.database** to be specified. This property must be specified for each server that is used in the **dbcdx.file.prep**, **dbcdx.file.open**, **dbcdx.file.vo**l, **dbcdx.file.prefix.***fsname* and **dbcdx.file.primary.server** properties. This property may be specified more than once, except *fsname* can not be duplicated.

dbcdx.file.server.*fsname***.serverport =** nnn

This property associates a server IP port number to the user defined *fsname*. It is an optional property for each **dbcdx.file.server.***fsname***.server** that is specified. *nnn* is the port number that is used for the initial TCP/IP connection to the FS Server. If this property is not specified, then a value of 9584 is used for non-encryption connections and 9585 is used for encryption connections. Otherwise, *nnn* can be a positive value that needs to match nport or eport in the FS Server configuration file for non-encrypted or encrypted connections respectively.

dbcdx.file.server.*fsname***.user =** *username*

This property associates the user name to the user defined *fsname*. It is an optional property for each **dbcdx.file.server.***fsname***.server** that is specified. If not specified, then a value of **DEFAULTUSER** is used. *username* must match a user name in the FS Server password configuration file as well as the password defined by **dbcdx.file.server.***fsname***.password**.

dbcdx.file.sharing = off

This property specifies single user mode. When this property is specified, all files are opened in exclusive mode.

dbcdx.file.source = *fsname*

dbcdx.file.source = *directory* [; *directory* …]

This property specifies the DB/C FS server or local directories to search when opening a program source file. If the value is a directory or a list of directories an attempt is made to locate that file in the directory or directories in the order specified. If the value is a DB/C FS server name, the database server will attempt to locate the file in the directory or directories that is controlled by the database configuration information.

dbcdx.file.updtfill = old

If this runtime property is specified, then an update with a list shorter than the record in the data file will not blank fill the remaining characters in the data file as in DB/C release 7.0.

dbcdx.file.vol.*volume* **=** *fsname*

dbcdx.file.vol.*volume* **=** *directory* [; *directory* …]

This property specifies the local directory or DB/C FS database associated with the *volume* that is specified. *volume* is case sensitive and must exactly match the volume specified on the open and prepare statements. If the value is a DB/C FS server name, then files will be created in the location that is controlled by the database configuration information. This property may be specified more than once, except *volume* can not be duplicated.

dbcdx.file.wttrunc = on

If this runtime property is specified, then a write or update operation with a list longer than the length specified in the file, ifile or afile definition will be truncated so as not to exceed the specified length as in DB/C release 7.0. This will inhibit I 716 errors from occurring when the list longer than the record length.

dbcdx.gui.clipboard.codepage= OEM

By default DB/C DX assumes that text being written to and read from the clipboard uses the Windows ANSI character set, which is essentially the same as ISO-8859-1 and is the character set generally used in Windows except in consoles. If this property is set, then clipboard load and store operations will assume that the OEM character set, also known as the IBM PC Extended Character Set, is being used. The OEM character set is used by Windows in consoles. The OEM character set varies somewhat between computers and depends on the code page ROM installed by the manufacturer.

`dbcdx.gui.ctlfont =` *font*

This property defines the font to be used for GUI controls. *font* is the font name without size or style information.

`dbcdx.gui.draw.stretchmode =` *digit*

This property defines the stretch mode used for DRAW stretching operations. This property is only applicable in Windows. The value of *digit* must be one of 1, 2, 3 or 4. These correspond to the folllowing windows stretching modes (1 is the default):
1 = STRETCH_DELETESCANS
2 = STRETCH_HALFTONE
3 = STRETCH_ANDSCANS
4 = STRETCH_ORSCANS

`dbcdx.gui.enterkey = old`

This property makes the enter key work as it did in DB/C DX release 11 and before. In panels and dialogs, the enter key acts like the tab key and no **NKEY** messages are sent to the queue when enter is pressed.

`dbcdx.gui.focus = old`

This property causes the GUI change control focus command to bring the window containing the control to the front of all windows.

`dbcdx.gui.listbox.insertbefore = old`

This property makes listboxes and dropboxes that are in insertorder behave as they did before DB/C DX release 14. This affects the insert behavior when an insertbefore change command is issued without any data. Prior to version fourteen, the insert point would be at the top of the list but each subsequent insert would move it down by one position. In version fourteen and newer, without this runtime option, the insert position will remain at the top.

`dbcdx.gui.pandlgscale =` *mmm* **/** *nnn*

This property specifies the ratio applied to all control sizes and positions in panel and dialog resources. It also affects the size of a dialog resource that is specified by the size parameter. It does not apply to the window size specified in the prepare statement, except if the pandlgscale keyword is specified, then the size of the window is scaled. Scaling does not apply to show positions, image variables, window positions, mouse positions, or any other GUI position or size.

`dbcdx.keyin.break = off`
`dbcdx.keyin.break = stop`

This property changes the default behavior of the DB/C DX runtime, which is to terminate if the break key is pressed. Setting this property to off prevents the termination of the DB/C DX runtime and the character pressed is processed normally. Setting this property to stop causes the break key to perform an implicit stop instruction when the break key is pressed. The break key is operating system dependent.

`dbcdx.keyin.case = upper`
`dbcdx.keyin.case = reverse`

This property specifies the translation of keyin character case. The following table describes the case of the resulting letter in different situations:

dbcdx.keyin.case	keyin control code	shifted keystroke	unshifted keystroke

none	***in**	upper-case	lower-case
none	***it**	lower-case	upper-case
upper	***in**	upper-case	upper-case
upper	***it**	lower-case	lower-case
reverse	none	lower-case	upper-case
reverse	***it**	upper-case	lower-case
any value	***uc**	upper-case	upper-case
any value	***lc**	lower-case	lower-case

`dbcdx.keyin.cancelkey =` *key*

key is one of: **none**, **tab**, **esc**, **^a** … **^z**, **F1** … **F20**. The default is **none**. This property specifies the keyin cancel key. Pressing the cancel key causes the input of a keyin variable to be erased.

`dbcdx.keyin.device =` *devicename*

This property specifies the input device or file associated with keyin operations.

`dbcdx.keyin.editmode = on`

If this runtime property is specified, when a variable is edited during keyin, the insert key may be used to toggle between insert and over strike modes.

`dbcdx.keyin.endkey = xkeys`

This property specifies that additional function keys will cause keyin to terminate. The default ending keys for a keyin instruction are **enter**, **F1**-**F20**, **up**, **down**, **left** and **right**. The additional function keys include **home**, **end**, **page up**, **page down**, **insert**, **delete**, **esc**, **tab** and **back tab**.

`dbcdx.keyin.fkeyshift = old`

This property applies to the Windows runtime and causes the pressing of shifted **F1**-**F10** to be the equivalent of pressing **F11**-**F20** respectively.

`dbcdx.keyin.handshake = dtr`
`dbcdx.keyin.handshake = cts`
`dbcdx.keyin.handshake = on`

This property applies to the Windows runtime. It determines the flow control when DB/C DX is executed through a communication port, which requires **dbcdx.display.termdef** to be defined. The default is **dtr**.

`dbcdx.keyin.ict=` *nnn*

This property applies only to Linux runtimes. In order to properly process multibyte key sequences from terminals, the runtime has a built in delay when it sees the first character of such a sequence. This property allows the user to adjust this delay. *nnn* is a number from 1 to 9 inclusive. It is the number of tenths-of-a-second to delay after the first byte of a potentially multibyte sequence to wait and see if more bytes are received. The default is 5.

`dbcdx.keyin.ikeys = on`

This property is valid for all versions of DB/C that are run on terminals with the appropriate keys. When this property is set, this property allows many special characters to be entered in keyin statements. Characters with decimal values 21 and 128–255 are valid when this property is set.

`dbcdx.keyin.interruptkey =` *key*

key is one of: **none**, **esc**, **^a** … **^z**, **F1** … **F20**. The default is **^z**. This property specifies the keyin interrupt key. Pressing the interrupt key causes an implicit stop instruction to occur.

`dbcdx.keyin.lowertranslate =` *translate-spec* [**`;`** *translate-spec*]

After the **dbcdx.keyin.translate** processing is done, if the case mapping processing is done. If ***lc** is in effect, then the **dbcdx.keyin.lowertranslate** translation is applied.

`dbcdx.keyin.translate =` *translate-spec* [**`;`** *translate-spec*]

The **dbcdx.keyin.translate** property is used to translate and validate entries during keyin.

When a keystroke is entered, the equivalent value of the character is used - as long as the value is non-zero. If the byte value of the key in the specification is zero, keyin of that character will not be permitted.

`dbcdx.keyin.uppertranslate =` *translate-spec* [`;` *translate-spec*]

If **`*`uc** is in effect, then the **dbcdx.keyin.uppertranslate** translation is applied.

`dbcdx.print.autoflush = on`

This property causes an implicit **`*`flush** command to be issued at the end of every print statement.

`dbcdx.print.cups = no`

This property tells the runtime that CUPS has not been installed on this machine. The default on Linux systems is to assume that CUPS is installed.

`dbcdx.print.destination = client`

This property causes print data to be directed to a print file or device on the client machine when Smart Client is being used. If this property is not specified, printing will occur on the server side.

`dbcdx.print.device =` *devicename*

This property sets the default print destination. *devicename* will be the print device if a print statement is issued without a SPLOPEN. If this option is not used, then the default print destination on Windows is the currently selected printer, and on Linux it is **/dev/lp**.

`dbcdx.print.language =` *language*

This property specifies the default print language used for splopen statements. Values of *language* are **pdf**, **pcl**, **pcl(noreset)**, **ps**, **native** and **none**. **native** causes printing to use the operating system print facilities. **none** causes basic line oriented print output. **native** is the default for the Windows GUI runtime (**dbc.exe**). **none** is the default for other runtimes.

`dbcdx.print.pcl.imageposition = old`

This property affects the relationship between the current cursor position and the drawing of an image. Images are drawn by default with the current cursor position at the upper left corner of the image. If this property is used, then for pcl only, images are drawn with the current cursor position at the lower left corner if the image.

`dbcdx.print.pcl.uom = old`

This property affects the unit-of-measure used for positioning output to PCL. That is, when using the **`*`p=h:v** print control and others that use pixels. The default method is to use PCL units. This is effectively the dpi as set by the Z option on the splopen statement. PCL units default to 300 per inch. If this property is specified, the unit of measure is 1/72 of an inch.

`dbcdx.print.translate =` *translate-spec* [`;` *translate-spec* …]

This property specifies the print translation table. In addition to the two types of values in *translatespec*, the print translate property allows a translate specification in the form *nnn*:**(***nnn***,** *nnn***,** …**)**.When this form is specified, the from translation character (the first *nnn*) is translated into a multiple character sequence made up of the characters in the parentheses.

SQL

DB/C DX provides for access to an SQL database management system. The DB/C program contains the sqlexec, sqlcode, and sqlmsg statements to access the SQL database.

In the Windows version of DB/C DX, the SQL interface is provided for databases that provide an ODBC driver.

In the Linux version of DB/C DX, the SQL interface is provided for databases that provide a unixODBC driver. The DB/C DX SQL interface for unixODBC is the file named **dbcsql.so**. This file must be located where the system loader can find it. This is typically in **/lib** or **/usr/lib**. An I 653 error will result if one of the sql statements is executed and the interface file cannot be found. For more information about unixODBC see **www.unixodbc.org**.

The sqlexec statement executes SQL statements. The sqlcode and sqlmsg statements provide the program with information about the success or failure the SQL statements executed with sqlexec.

The three forms of the sqlexec statement are:

`sqlexec` *charexp*
`sqlexec` *charexp* **`from`** *from-list*
`sqlexec` *charexp* **`into`** *into-list*

The *charexp* operand contains the actual SQL statement. This operand can be either a character literal or a character variable. Within the actual SQL statement, expressions of the form **:***n*, where *n* is an integer, are replaced with the corresponding variable in the from-list. For example, in the expression:

```
sqlexec "SELECT : 1 USING C1 FROM TABLE" from var1
```

:1 is replaced with the value in **var1** before the statement is passed to the SQL server. Care should be taken that variables in the *into-list* are large enough to hold the results, and that variables in the *from-list* are not too large for the table. Of special note, an SQL variable declared as DECIMAL(3,2) is the same size as a DB/C variable defined as NUM 1.2.

The sqlexec statement affects flags as follows. If an SQL statement returns an error condition, then the less flag is set and the over and equal flags are cleared. If the SQL statement is successful, then the equal flag is set and the less and over flags are cleared. If the SQL statement returns no data, the over flag is set and the less and equal flags are cleared. This will occur when there is no data from an SQL FETCH statement.

The sqlmsg statement places any message associated with the most recently executed SQL statement into a character variable. The syntax is:

`sqlmsg` *charvar*

The sqlcode statement places the code returned by the SQL server into a numeric variable. The syntax is:

`sqlcode` *numvar*

Connect

Connections to the SQL server are made with the SQL CONNECT statement. Here is an example:

```
sqlexec "CONNECT SERVER=dbmain, NAME=user1, PASSWORD=pswd"
```

These five keywords are accepted: **SERVER**, **NAME**, **PASSWORD**, **CONNECTION** and **STRING**. Each of the keywords is followed by an equal sign and a value. The keywords may be in any order. The **SERVER** is required. The others are optional. **SERVER** is the database or dataset name. **NAME** and **PASSWORD** are the user and password. **CONNECTION** specifies the connection name. The connection name may be up to 20 characters. **STRING** is an ODBC driver specific string of characters.

The **CONNECTION** keyword is unnecessary if only one database will be accessed. It is useful when concurrent access will be done to two or more databases. The connection clause is **CONNECTION** *connection*, where *connection* is the value specified with the **CONNECT** keyword in the connection statement. The connection clause is specified in each SQL statement, just like other clauses in the SQL language. Here is an example:

```
SQLEXEC "CONNECT SERVER=a, NAME=user1, PASSWORD=pswd, CONNECTION=C1"
SQLEXEC "DELETE CONNECTION C1 FROM TABLE1 WHERE NAME='JIM'"
```

Select and Fetch

When retrieving results from an SQL server, it is necessary to use cursors. This is because an SQL SELECT statement can result in multiple rows of data that must be fetched one at a time into DB/C program variables. To use a cursor, the **USING** *cursor-name* clause must be in the SQL statement. The *cursor-name* may be up to 20 characters long. Up to fifty different cursor-names may be declared and in used.
A *cursor-name* may be redeclared with a new SELECT statement to have a new meaning.

There are two options for cursors that may be set in the statement. The first is **SCROLL**, which determines membership in the cursor. The second is **LOCK**, which determines how concurrency control is implemented. These options are not available in certain SQL implementations. Options are specified like this:

```
sqlexec "SELECT OPTIONS(SCROLL=FORWARD, LOCK=FETCH) FROM TABLE ..."
```

The **OPTIONS** keyword must be immediately after the connection clause if it exists, or immediately after the **SELECT** keyword if a connection clause doesn't exist. Either or both options can be missing and the keywords are case insensitive.

The **SCROLL** options are:

FORWARD	The rows in the cursor can only be retrieved in order.
KEYSET	The rows in the cursor are determined at SELECT time, but the data in them can change.
DYNAMIC	The rows and the data in the cursor can change.

The **LOCK** options are:

READONLY	The cursor is read only.
OPTBYTIME	Changes are detected using special concurrency columns like TIMESTAMP and ROWID.
OPTBYVAL	Changes are detected by comparing values in selected columns.
FETCH	Rows are locked when they are fetched.

Rows are retrieved using the SQL FETCH statement. Here is the syntax of a FETCH statement that fetches the next row in the cursor into the list of DB/C variables in the into-list:

sqlexec "FETCH USING *cursor-name***" into** *into-list*

The FETCH statement may also specify a row to retrieve. The syntax for this is as follows:

sqlexec "FETCH *row-spec* **USING** *cursor-name***" into** *into-list*

The values for row-spec are:

NEXT	gets the next row
PRIOR	gets the previous row
PREV	gets the previous row
PREVIOUS	gets the previous row
FIRST	gets the first row
LAST	gets the last row
RELATIVE *n*	gets the *n*th row from the current position
ABSOLUTE *n*	gets the *n*th row in the cursor

Update and Delete

Two other important SQL statements are UPDATE and DELETE. Each has two distinct modes - with or without a cursor. Without a cursor, the effect of the SQL UPDATE and DELETE statements applies to multiple rows of a table. For example:

```
sqlexec "UPDATE PRODUCTS SET PRICE = 12.50 WHERE PRICE = 0"
```

This would set all rows with zero price to have price of 12.50.

The SQL UPDATE and DELETE statements may also performed through cursors. When updating or deleting through a cursor, the current row of the cursor is updated or deleted. For example:

```
sqlexec "DELETE FROM PRODUCTS WHERE CURRENT USING CURSOR2"
```

This SQL statement will delete the most recently retrieved row using the cursor **CURSOR2**.

In the case of UPDATE through a cursor, the lock option on the SELECT statement is important. Updates will not be allowed if the cursor is a READONLY cursor. Updates are guaranteed to succeed if the LOCK=FETCH option is specified. Depending on which SQL product is being used, you may have to use the statements like BEGIN TRANSACTION and END TRANSACTION, around the FETCH and UPDATE statements. If the LOCK option is set to OPTBYVAL or OPTBYTIME, values will be compared before the update takes place. If they are different than what was fetched, the update will fail.

NULL values

DB/C DX provides support for NULL values. A NULL value is sent to the SQL server if the DB/C variable in the *from-list* has the NULL value. The DB/C variable in an *into-list* will be set to the NULL value if a NULL value is returned. The DB/C setnull statement is used to change the value of a character or numeric variable to NULL. The DB/C isnull operator is used to indicate the NULL or non-NULL status of a character or numeric variable.

It is also possible to use the NULL SQL keyword to pass NULL values to the SQL server. Here is an example:

```
sqlexec "INSERT INTO TABLE1 VALUES (NULL)"
```

Communications

The communication statements (comopen, send, recv, etc.) provide a generalized means of sending messages to and from other devices or programs. Serial port communications and TCP/IP communications are supported in DB/C DX.

Serial Communications

For Windows, up to 99 serial ports are supported. The device name (*device* below) is **com1**, **com2**, ... **com99**.

For Linux, all **/dev/***xxx* devices that are connected to serial ports are supported. The Linux communications interface does not support the comctl statement.

To open a serial port, the comopen statement contains the serial port parameters. These parameters are:

`SERIAL` *device* **:** *speed, parity, bits, stopbits, instart, inend, outstart, outend, ignore, length*

speed is the baud rate. Valid values are **3**, **6**, **12**, **24**, **48**, **96**, **192**, **384**, **576**, and **1152**. These values represent 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, and 115200 baud, respectively. If this value is not specified, then the current operating system setting is used.

parity is one of **n**, **o**, or **e**. These represent **none**, **odd**, and **even**, respectively. If this value is not specified, then the current operating system setting is used.

bits is the number of bits per character. Valid values are **5**, **6**, **7**, or **8**. If this value is not specified, then the current operating system setting is used.

stopbits is the number of stop bits per character. Valid values are **1** or **2**. If this value is not specified, then the current operating system setting is used.

The *instart*, *inend*, *outstart*, *outend*, and *ignore* parameters consist of the standard ASCII printable characters (hex 20 through 7E) except for **^** and ****. When a **^** is en countered, the following character must be one of **a** through **z**. This combination signifies the control characters Ctrl-A through Ctrl-Z (hex 01 through 1A). Whenever a **** is encountered, the following three characters must be decimal digits. These three digits signify the decimal value of the character. Values from 000 through 255 may be specified (for example, \000 is hex 00 and \127 is hex 7F). Use \\ and **\^** for the characters **** and **^** respectively. Specifying a parameter as null **,,** causes that parameter to be unsupported (no default character is assumed).

instart is the input message start character sequence. This sequence of characters marks the beginning of the message that will be received by the recv statement. When the message is put into the DB/C program variables, this character sequence is not included. If this value is not specified, no default character is assumed.

inend is the input message termination character sequence. This sequence of characters marks the end of the message that will be received by the recv statement. When the message is put into the DB/C program variables, this character sequence is not included. If this value is not specified, the default character **^m** is assumed.

outstart is the output message start character sequence. This sequence of characters is appended to the beginning of the message being sent by the send statement. If this value is not specified, no default character is assumed.

outend is the output message termination character sequence. This sequence of characters is appended to the end of the message being sent by the send statement. If this value is not specified, the default characters **^m^j** are assumed.

ignore is the list of input characters to be ignored. All characters specified will be ignored. If this value is not specified, the default character **^j** is assumed. length is the input message length parameter. When the specified number of characters is received, a message is considered to be complete and the recv statement will be satisfied. When zero is specified as the length, then messages are terminated by the *inend* character sequence. The default is zero. If both *length* and *inend* are defined, then first condition to occur causes the message to be considered complete.

The comctl operation is typically used to control serial communications and to return information for TCP communications. comctl may also be used to return the error condition on a commuication failure. Use "GETERROR" as input after receiving an I 753 error. Use "GETSENDERROR" as input if comtst returns with equal and eos set. Use "GETRECVERROR" as input if comtst returns with less and over set. Output will contain an error message regarding the error or "TIMEOUT" if the send or recv statement timed out.

The comctl operation controls the settings of control signals on serial ports. The character variable operand may contain zero or more of the following characters with these meanings:

`R`	the RTS line will be set high
`r`	the RTS line will be set low
`T`	the DTR line will be set high
`t`	the DTR line will be set low
`C`	characters will be sent only when CTS is high
`c`	characters will be sent regardless of CTS
`S`	characters will be sent only when DSR is high
`s`	characters will be sent regardless of DSR
`L`	characters will be sent only when DCD is high
`l`	characters will be sent regardless of DCD
`X`	enable XON/XOFF processing
`x`	disable XON/XOFF processing

On return, the comctl statement will change the character variable operand to contain a length four string containing upper or lower case characters these characters with these meanings:

`C`	the CTS line is high or
`c`	the CTS line is low, followed by
`S`	the DSR line is high or
`s`	the DSR line is low, followed by
`L`	the DCD line is high or
`l`	the DCD line is low, followed by
`B`	the RI line is high or
`b`	the RI line is low

The default values in effect after a comopen are: **RTcSlx**. Not all control signals are available on all operating systems.

For serial ports, the comctl statement also supports the following keywords:

`INSTART=`*value*	the input message start character string
`INEND=`*value*	the input message termination character sequence
`OUTSTART=`*value*	the output message start character string
`OUTEND=`*value*	the output message termination character sequence
`IGNORE=`*value*	the list of input characters to be ignored
`LENGTH=`*value*	the input message length parameter

Setting of values is the same as comopen. On return, the comctl statement will change the character variable operand to contain the input keyword and the value of the keyword prior to the comctl. This string returned may be used as input to comctl to reset the value back to the previous setting.

TCP/IP

The TCP/IP implementation of the communications statements gives the DB/C programmer direct access to the two most important Internet protocols, TCP and UDP, and to the DNS service.

TCP communications takes place between two endpoints (also referred to as sockets). Endpoints are uniquely determined by their IP address and port number. An IP address is unique to the machine the endpoint is on, and is specified in one of two ways, for example:

```
216.58.161.16
ftp.dbcsoftware.com
```

These addresses are actually the same. The first is the four byte address used by the IP protocol. It is typically written like *nnn***.***nnn***.***nnn***.***nnn* where each *nnn* is a value between 0 and 255, inclusive. The second address is in domain name format. The purpose of a Domain Name Server (DNS) is to convert a domain name to its IP address.

The port number element of an endpoint is a value from 1 to 65,536.

Thus for any given machine with a single IP address, there are 65,536 unique endpoints. In practice, most operating systems allow a much smaller number of endpoints to be active at any one time.

Port numbers are broken down into ranges that are available for different kinds of services. Ports 1 to 1023 are the "well known port numbers". For example, port 23 is the telnet port. This means that telnet server software listens for telnet clients on port 23.

Port numbers 1024-5000 are the "ephemeral port numbers". These port numbers are assigned to applications that request use of an endpoint without specifying a port number. This behavior is typical of a client application. These endpoints tend to exist for a limited time, hence the name ephemeral.

Port numbers above 5000 are available to server applications by specific request. Server programs must use endpoints with predetermined addresses so that clients know where and how to connect with servers.

The comopen statement is used to open a TCP connection. There are two ways to open a TCP connection, one as the client and the other as the server. Here is an example of how to open the client end of a connection:

```
cfile    comfile
         comopen cfile, "TCPCLIENT ftp.dbcsoftware.com 21"
```

Just as with other types of communications operations, a comfile variable is used as the anchor for a session or connection. The **TCPCLIENT** keyword in the last operand specifies that the client connection is to be opened. One or more blanks separate each of the three values in the last operand. The second value is the IP address or DNS name.

In the example, **ftp.dbcsoftware.com** is the domain name. The third value is the port number.

After the connection is made, the endpoint on the client end of the connection will have the IP address of the client machine and an ephemeral port number that the operating system assigns to it. You can use the comctl statement to retrieve the IP address of an endpoint, like this:

```
ctl      char 50
cfile    comfile
         comopen cfile, "TCPCLIENT 216.58.161.16 21"
         move "GETLOCALADDR" to ctl
         comctl cfile, ctl
```

The local address and port are moved into the variable named **ctl**.

Here is an example of opening a TCP server connection:

```
cfile    comfile
         comopen cfile, "TCPSERVER 21"
```

When **TCPSERVER** is used in the comopen statement, a server endpoint is opened waiting for incoming connections. The IP address is that of the local machine, and the port number that is specified in comopen statement. In this example, 21 is the port number. When another application connects to this endpoint, the connection is made on another port on the server computer. This insures that another comopen using the original port will continue to wait for connections on the original port. This means that after a connection is completed the comctl statement will return a different port number than the port number specified in the comopen statement.

IPv6 is supported. For **TCPCLIENT**, both IPv4 and/or IPv6 connections are attempted depending on the operating system. If **TCPSERVER4** is specified, the server port opened will be for IPv4. If **TCPSERVER6** is specified, the server port opened will be for IPv6. If **TCPSERVER** is specified, either IPv4 or IPv6 may be selected depending on the configuration of the computer.

Here is an example of opening multiple server connections on a port:

```
cfile1   comfile
cfile2   comfile
cfile3   comfile
         comopen cfile1, "TCPSERVER 9999"
         comopen cfile2, "TCPSERVER 9999"
         comopen cfile3, "TCPSERVER 9999"
```

Messages are transmitted using the send and recv statements. In the case of both TCPCLIENT and TCPSERVER, the send and recv statements will only be satisfied after the connections are actually made. The success of a comopen statement with the TCPCLIENT keyword means that the socket was successfully opened.

The syntax of the send and recv statements is as follows:

send *comfile*, *numvar*; *list*
recv *comfile*, *numvar*; *list*

The timeout value is specified by the *numvar* operand and is specified in seconds. A value of zero means an immediate timeout. A negative value means no timeout.

TCP is a stream based protocol. A send statement will set the comfile variable to the send completed state when the requested list has been sent or a timeout or error occurs. The recv statement is slightly different. The comtst statement will indicate that a recv statement has successfully completed as soon as any amount of data is available. This means that any use of message boundaries must be provided for by the application.

UDP is a connectionless, message based protocol. Here is an example of how to open an endpoint for UDP communications:

```
cfile    comfile
         comopen cfile, "UDP 8000"
```

This comopen statement opens an endpoint with the IP address of the local machine and the port number specified by the value after the UDP identifier. The comctl statement returns the IP address and port number the same way as it does for TCP.

A send or recv statement on a UDP endpoint results in a whole message being sent or received. In order to specify the destination for sending a message, the comctl statement is used. Here is an example:

```
cfile    comfile
ctl      char 50
msg      init "hello"
time     int "-1"
         comopen cfile, "UDP 11005"
         move "SETSENDADDR 207.208.157.42 10005" to ctl
         comctl cfile, ctl
         send cfile, time; msg
         comwait cfile
         comtst cfile
         if (equal and not eos)
                  display "success"
         else
                  display "error"
         endif
```

This example will cause UDP messages to be sent from local port 11005 to port 10005 on the machine at 207.208.157.42.

After a UDP message has been received, the comctl statement can be used to find out where it came from.

In the following example, if the recv is successful, the IP address and port number of the program that sent the message will be placed in the variable **ctl**:

```
cfile    comfile
ctl      char 50
msg      char 5
time     int "-1"
         comopen cfile, "UDP 10005"
         recv cfile, time; msg
         comwait cfile
         comtst cfile
         if (less and not over)
                  display "success"
                  move "GETRECVADDR" to ctl
                  comctl cfile, ctl
         else
                  display "error"
         endif
```

Writing Portable Programs

When certain rules are followed, DB/C programs can be completely independent of the operating system. More specifically, any compiled program (**.dbc** file) or source chain file will run without modification on any version of DB/C DX. The following is a list of the rules that must be followed to achieve complete operating system independence:

1. The rollout and execute statements must run only DB/C DX utility programs.

2. All file names (in open statements, utilities, etc.) must be in standard DB/C file name format or in a Datapoint file name format using the **dbcdx.file.compat** runtime property.

The format of a standard DB/C file name is:

> *name*.*ext*:*volume*
> *name* is one to eight characters long
> *ext* is one to three characters long
> *volume* is one to eight characters long

The standard DB/C file name is not a valid file name for any operating system.

The format of a non-standard file name is any format other than a standard DB/C file name, but still a valid file name format for the runtime operating system. Some operating systems differentiate between upper and lower-case letters in file names. In other operating systems, no distinction is made.

3. Use only DB/C-type files. Use of text, native, or binary type files may cause portability problems. When other than DB/C files are used, certain statements (such as fposit) may work differently on different versions of DB/C.

4. The utilities should be renamed to names that are not duplicates of built-in commands in the operating system. Suggested names are: dcopy, dcreate, ddelete, drename, and dsort.

5. Splopen of devices must not be used. Some operating systems do not support opening devices as files.

6. Any other device dependencies (such as octal characters in print statements) should not be used.

Windows Considerations

System Requirements

DB/C DX for Windows requires an x86-based 64-bit version of Windows that is Windows 7 or newer. The Windows runtime interpreter is **dbc.exe**.

Keystroke Definition

The keyboard usually has F1 through F12 keys and a backspace key. The break key is the Ctrl-C or the Ctrl-Break key. Specifying the **dbcdx.keyin.break=off** runtime property will enable the Ctrl-C character to pass through and Crl-Break will be ignored.

File Names

Standard Windows file names are of the form:

[*drv* `:`] [`\`*dir*...] *name* [`.`*ext*]

The *drv* and *dir* specifications constitute the path name. The *drv* specification is one character. A file name like **a:xyz** will be considered a non-standard file name (file **xyz** on drive **a**). A file name like **ab:xyz** will be considered a standard DB/C file name (file **ab** on volume **xyz**).

No distinction is made between lower and upper-case letters in file names. For example, **xyz.txt** and **xyz.TXT** are the same file.

Characters

The **-t** command line option and the text option in file, ifile, and afile declarations are implemented with the ASCII character set.

The end-of-record specifier is CR, LF (carriage return followed by line feed, or hex values 0D,0A).

Records deleted by the delete statement are replaced with DEL (hex 7F).

Some older programs may use the Ctrl-Z character to signal end-of-file, so DB/C will recognize existing Ctrl-Z (hex 1A) end-of-file marks.

Size of the Screen

The runtime properties

```
dbcdx.display.columns=number
dbcdx.display.lines=number
```

specify the number of columns and lines available for keyin and display. The default is 80 by 25.

Printing

The default system printer is LPT1.

The default system printer may be used by specifying splopen with a printer name of DEFAULT. The size of each print dot is one printer resolution dot. This may be modified by enclosing the requested size in parentheses after the printer name. For example:

```
          splopen "DEFAULT(300)"
```

causes the default system printer to be opened with a resolution of 300 dots per inch. Any printer name defined by the Windows Print Manager is a valid name for the splopen statement.

There are five output types when printing with the Windows version of DB/C DX. They are: FILE, RECORD, DEVICE, COOKED, and RAW.

The FILE output type means that print output is sent to a file in the file system. Pixel level and graphical print controls may or may not be supported depending on the output format.

The RECORD output type is almost the same as FILE. It is used only when the output format is CC (see below). The output file is a runtime operating system text file. The expectation is that the record separaters will not be interpreted by the printer. Pixel level and graphical print controls are not supported by this output type.

The DEVICE output type means that printing goes directly to an attached device, for example **LPT1**. Pixel level and graphical print controls may or may not be supported depending on the output format.

The COOKED output type means that the print commands are passed through the Windows printing API to the Windows printer driver. Pixel level and graphical print controls are supported. This is the most portable Windows printer output type.

The RAW output type causes output to be sent to the printer unmodified. Pixel level and graphical print controls may or may not be supported depending on the output format.

There are five output formats. They are: Device Control Characters (DC), Carriage Control Characters (CC), PDF, PCL and PostScript.

The DC output format uses standard ASCII device control characters for form feed, line feed, and carriage return. Pixel level and graphical print controls are not supported by this output format.

The CC output format causes a carriage control character to be put in the first character position of each record. The carriage control characters are: "1" (top of form), "+" (no vertical skip) and blank (next line). If the output is to a file it will be a runtime operating system text file. Pixel level and graphical print controls are not supported by this output format.

The PDF output format produces Portable Document Format output. Pixel level and graphical print controls are supported by this output format.

The PCL output format produces Printer Control Language output. Pixel level and graphical print controls are supported by this output format.

The PostScript output format produces PostScript output. Pixel level and graphical print controls are supported by this output format.

When the splopen statement is executed, the output type is determined by examining the file or device name. The splopen options may override the output type.

If the print destination name is **PRN**, **AUX**, **NUL**, **LPT***n*, **LPT***nn*, **COM***n*, or **COM***nn* where *n* is a decimal digit, then the destination type is DEVICE. If the name is not DEFAULT and the name is not a match for anything in the Windows Printers folder then the destination type is FILE. Otherwise, if the name contains parentheses as described above then the output type is COOKED, otherwise the output type is RAW.

The **L** splopen option may alter the output type and format. The following table shows the resulting print output type for various combinations of destination names and **L** options.

	named printer	**named printer (dpi)**	**print file**
none	COOKED	COOKED	RECORD
L=NONE	RAW	ERROR P 411	RECORD
L=NATIVE	COOKED	COOKED	RECORD
L=PDF	RAW (note 1)	COOKED (note 2)	FILE
L=PCL	RAW (note 3)	COOKED (note 2)	FILE
L=PS	RAW (note 4)	COOKED (note 2)	FILE

notes: 1. Raw PDF is sent to the printer
2. Print languages don't support **(dpi),** forced to **L=NATIVE**

3. Raw PCL is sent to the printer
4. Raw PS is sent to the printer

Additional Information about SPLOPEN Options

B=*bin*	This option only works if the output type is COOKED. Each printer driver has its own set of paper bin options that it will recognize. Use the getpaperbins statement to obtain a list of available bins.
I	If output is to a file, and the **SUBMIT** option of the splopen statement is used, then no trailing form feed will be sent to the printer. Otherwise it is ignored.
J	This option will set the output type to COOKED, exactly as if **L=NATIVE** was specified. A standard Windows print dialog box will be displayed.
L=*lang*	This is the print output language option. This option sets the output format and may affect the output type. See the chart on the preceding page for the combinations of destination name and **L** option.
N	This option works with COOKED output type only.
O=*orientation*	This option works with the COOKED output type. It will work with FILE, RAW, and DEVICE output types if the output format is PCL, PDF, or PostScript.
S=*size*	If the output type is COOKED then size must be one that is supported by the printer driver of the printer being opened by the splopen statement. It will be used for RAW, DEVICE, and FILE output types if the output format is PCL, PDF, or PostScript. If the output format is PDF, PCL, or PostScript then valid values for size are: **LETTER** **LEGAL** **COMPUTER** **A3** **A4** **B4** **B5**
W	This option works only with COOKED output type.
Y=*text*	This option sets the document name as it appears in the printer queue window. It only works with COOKED and RAW output types.

Additional Information about SPLCLOSE Options

SUBMIT [=*string*] This option assumes that the output is to a file. If *string* is not specified the file will be submitted to the default printer. Otherwise the file will be submitted to the printer named *string*. If *string* is not recognized by the operating system a P 452 error will occur.

CLOCK Return Values

clock version returns **"Windows"**.

clock port returns **"001"**. This may be altered by the **dbcdx.clock.port** runtime property.

clock user returns the login name.

The *pp* characters returned by clock timestamp will correctly report hundredths of a second.

PAUSE and TRAP

The pause, trap timeout and trap timestamp statements support hundredths of a second.

Linux Considerations

This chapter contains information about Linux support for DB/C DX. The character mode interface is the only user interface that is supported directly on the computer console or via TTY (e.g. **ssh**) interface. The GUI features of DB/C DX are supported by using Smart Client.

Keystroke and Display Definitions

The break key is the interrupt control character (**stty intr**). Specifying the **dbcdx.keyin.break=off** runtime property will cause the interrupt control character to be ignored. Some Linux systems define Ctrl-Z as the suspend control character (**stty susp**). This will conflict with the DB/C default interrupt key. This conflict can be resolved using the **dbcdx.keyin.interrupt** runtime property or the Unix stty(1) command.

If the **dbcdx.display.termdef** runtime property is used, it controls all keystroke definitions and display functions. If it is not used, then the terminfo database is used.

The following functions are available if the respective terminfo entry is defined:

Function	terminfo code
initialize terminal	**scmcup**, **smkx**
screen erased with background color	**bce**
number of **lines**	**lines**
number of columns	**cols**
`*p=`*h*`:`*v*, `*n`, `*c`, `*l`	**cup**, **hpa**, **vpa**
`*es`	**clear**
`*ef`	**ed**
`*el`	**el**
underline cursor on	**cnorm**
block cursor on	**cvvis**
cursor off	**civis**
`*r`	**ind**
`*rd`	**ri**
`*setswtb`	**csr**
`*dellin`	**dll**
`*inslin`	**il1**
`*delchr`	**dch1**
`*inschr`	**ich1**
`*revon`, `*hon`	**smso**
`*revoff`	**rmso**
`*ulon`	**smul**
`*uloff`	**rmul**
`*dion`, `*v2lon`, `*boldon`	**bold**
`*blinkon`	**blink**

Function	terminfo code
`*dimon`	**dim**
`*dioff`, `*boldoff`, `*blinkoff`, `*dimoff`	**sgr0**
`*b`	**bel**
`*pon`	**mc5**
`*poff`	**mc4**
`*hln`, `*vln`, and other line control codes	**acsc**, **smacs**, **rmacs**
`*color`	**max_colors**, **setf**, **setaf**
`*bgcolor`	**max_colors**, **max_pairs**, **setb**, **setab**
F1, F2, ... F20 keystrokes	**kf1**, **kf2**, ... **kf20**
page up keystroke	**kpp**
page down keystroke	**knp**
right keystroke	**kcuf1**
left keystroke	**kcub1**
up keystroke	**kcuu1**
down keystroke	**kcud1**
insert keystroke	**kich1**
delete keystroke	**kdch1**
home keystroke	**khome**
end keystroke	**kend**
backspace keystroke (note 1)	**kbs**
back tab keystroke	**kcbt**

notes: 1. If **kbs** is not defined in the termdef, then Ctrl-H is the backspace keystroke.
2. Ctrl-J and Ctrl-M are used for the enter keystroke.
3. Ctrl-I is used for the tab keystroke.
4. Ctrl-[is used for the escape keystroke.

File Names

In Linux, file names are of the form:

[`/`*dir*...] *name* [`.`*ext*]

The directory specification constitutes the path name.

There is a distinction between lower and upper-case letters in file names. For example, **xyz.txt** is not the same file as **xyz.TXT**.

Opening file names that have no extension is supported in Linux. If the file name specified in the open statement ends in a period, then the period is removed and no extension is added. If the file name ends in two periods, then one period is removed.

Characters

The text option in file and ifile declarations is implemented with the ASCII character set.

The end-of-record specifier is LF (line feed, or hex 0A).

Records deleted by the delete statement are replaced with DEL (hex 7F).

There is no end-of-file character. The end-of-file position is maintained by the operating system.

Tab character expansion on read is not supported. Any tab character encountered is translated to a single blank character.

Printing

The default system printer is **/dev/lp**.

There are four print output types. They are: FILE, DEVICE, PIPE, and CUPS (Common Unix Printing System).

The FILE output type means that print output is sent to a file in the file system. Pixel level and graphical print controls may or may not be supported depending on the output format.

The DEVICE output type means that printing goes directly to an attached device, for example, **/dev/lp**. Pixel level and graphical print controls may or may not be supported depending on the output format.

The PIPE output type assumes that the first operand of SPLOPEN is a command line which is started as a separate process. Print output is sent as standard input to this process

The CUPS output type means that the print job will be submitted to CUPS.

There are five output formats. They are: Device Control Characters (DC), Carriage Control Characters (CC), PDF. PCL and PostScript.

The DC output format uses standard ASCII device control characters for form feed, line feed, and carriage return. Pixel level and graphical print controls are not supported by this output format.

The CC output format causes a carriage control character to be put in the first character position of each record. The carriage control characters are: "1" (top of form), "+" (no vertical skip) and blank (next line). If the output is to a file it will be a runtime operating system text file. Pixel level and graphical print controls are not supported by this output format.

The PDF output format produces Portable Document Format output. Pixel level and graphical print controls are supported by this output format.

The PCL output format produces Printer Control Language output. Pixel level and graphical print controls are supported by this output format.

The PostScript output format produces PostScript output. Pixel level and graphical print controls are supported by this output format.

When SPLOPEN is executed, the output type is determined by examining the file or device name. The SPLOPEN options may override the output type.

If the **P** option is specified then the output type is PIPE. If the first five characters are **/dev/** then the output type is DEVICE. If the destination name begins with **lp** or **lpr** then the output type is PIPE. If CUPS is installed and the destination name is recognized by the CUPS daemon as a CUPS destination, then the output type is CUPS. Otherwise the output type is FILE.

Additional Information about SPLOPEN Options

`B=`*bin*	This option is only applicable to the CUPS output type. Use the getpaperbins statement to obtain a list of available bins.
`BANNER`	This option is only applicable to the CUPS output type. It causes a banner page to print.
`I`	This option is used only if output is to a file, CUPS is not installed, and the SUBMIT option of SPLOPEN is used. Otherwise it is ignored.
`J`	This option is ignored on Linux systems, unless running with Smart Client.

`L` This option works for all four output types. When using CUPS output, the CUPS system may be configured to pass the output through a filter. Refer to CUPS documentation for details.

`N` This option works with FILE output in conjunction with the SUBMIT option of the splclose statement. It also works with CUPS output.

`O=`*orientation* This option works with the CUPS output type. It will work with FILE, PIPE, and DEVICE output types if the output format is PCL, PDF, or PostScript. It will work with FILE output if the SUBMIT option of the splclose statement specifies a CUPS destination.

`P` This option will open a pipe to the executable named in the first operand. If the first operand of SPLOPEN begins with **lp** or **lpr** then this option is assumed. The first operand can contain arguments to the command. The splclose statement closes the pipe. For example:

```
splopen "lp -ddest -n2", "p"
print "Printing 2 copies to destination dest"
splclose
```

`S=`*size* If the output type is CUPS then size must be one that is supported by the printer configuration of the print destination being opened by the splopen statement. This option will work with the FILE output type if the SUBMIT option of the SPLCLOSE statement specifies a CUPS destination. This option will work for FILE and DEVICE output types if the output format is PCL, PDF, or PostScript. If the output type is PDF, PCL, or PostScript then values for size are:

```
LETTER
LEGAL
COMPUTER
A3
A4
B4
B5
```

`W` This option works only with CUPS output type, or FILE output type if the SUBMIT option of the splclose statement specifies a CUPS destination.

`Y=`*text* This option sets the job name. It only works with the CUPS output type.

Additional Information about SPLCLOSE Options

`SUBMIT` [=*string*] This option assumes that the output was to a file. The file will be submitted to the **lpr** or **lp** command. If string is supplied, string will be used as the destination name.

CLOCK Return Values

clock version returns **"UNIX"**.

clock port returns a length three string that contains the terminal number. For example, if the terminal is **/dev/tty02**, then clock port returns **"002"**. The string will always be filled with zeros on the left. The string may contain alphabetic characters in cases where the port is prefixed with **/dev/tty** and less than three digits on the end. For example, **/dev/ttya1** returns **"0a1"**. This may be altered by the **dbcdx.clock.port** runtime property.

clock user returns the user ID from log in.

The *pp* characters returned by clock timestamp will correctly report hundredths of a second.

PAUSE and TRAP

The pause, trap timeout and trap timestamp statements support hundredths of a second.

Sound

The sound statement is implemented like the beep statement. The operands for the pitch and the duration of the tone are ignored.

Click

The ***click** control code of the display statement and the ***clickon** control code of the keyin statement are ignored.

C Interface

Writing ccall and cverb Routines

The ccall and cverb functions, usually written in the C language, are called from the DB/C DX source program. A ccall routine is called by executing a ccall statement in a DB/C program. A cverb routine is called by executing a verb defined as a user defined verb with the cverb statement.

A ccall statement has the following format:

> *label* **`ccall`** *charexp prep list*
>
> *charexp* is the first operand
> *list* is a list of variables or literals

The ccall function is compiled and linked into the DB/C DX runtime. The name of the C function is ccall. The linkage to `ccall` is:

```
void ccall(unsigned char *name, unsigned char **list)
```

The cverb function is similar. The linkage is:

```
void cverb(unsigned char *name, unsigned char **list)
```

The parameter **name** is a pointer to the variable that is the first operand of the ccall operation or is the name of the cverb. The parameter **list** is a pointer to an array of pointers that each point to the variables that make up the list. The last pointer in the array of pointers has a null value.

The information necessary to compile and link the ccall function is different for each version of DB/C DX. The **dxreadme.txt** file supplied on the distribution media contains this information.

The following paragraphs describe the internal representation of variables.

There are two kinds of character variables: small and large. Small character variables are defined as char 127 or smaller. Large character variables are defined char 128 and larger.

A small character variable is stored as a string of bytes that is three bytes longer than the size of the variable. Three header bytes precede the actual characters that make up the character variable. Each of these header bytes is considered to be an unsigned character that contains a value between 0 and 127, inclusive. The first byte is the form pointer. The second byte is the length pointer. The third byte is the maximum length (or physical length) of the variable. The maximum length does not include the three header bytes. Because the value of the form pointer can only contain a value between 0 and 127, the high-order bit is always 0. This fact is used to distinguish small character variables from other types of variables.

A large character variable is stored as a string of bytes that is seven bytes longer than the variable. Seven header bytes precede the actual characters that make up the character variable. The first byte is decimal 240. This byte signifies that what follows is a large character variable. The second and third bytes are the form pointer in ll,hh format. The fourth and fifth bytes are the logical length pointer in ll,hh format. The sixth and seventh bytes are the maximum length (or physical length) of the variable in ll,hh format. The maximum length does not include the seven header bytes.

A form variable is stored as a string of bytes that is one byte longer than the variable. The header byte precedes the actual characters that make up the numeric variable. In the header byte, the high-order bit always contains the value 1, the next two bits contain the value 0 and the low five bits are the length of the numeric variable, not including the header byte.

An integer variable is stored as six bytes. The first byte is decimal 252. The second byte contains the format of the variable. This format is the number of decimal digits to which the value will be expanded when it is accessed for any operation other than a calculation. The following four bytes make up the 32-bit integer value stored in the correct format for the machine on which DB/C DX is running. Note that these bytes may not be aligned correctly for machines that are unable to do operations on an odd address.

A float variable is stored as ten bytes. The first byte has a value of 248, 249, 250, or 251. The low-order two bits of the first byte plus the eight bits of the second byte form the 10-bit format of the variable. The high-order five bits define the number of digits to the left of the decimal point and the low-order five bits define the number of digits to the right of the decimal point. The remaining eight bytes contain the double float value in the correct format for the machine on which DB/C DX is running. Note that these bytes may not be aligned correctly for some machines.

Numeric and character literals are stored as a string of bytes that is 2 bytes longer than the variable. Two header bytes precede the actual characters that make up the literal. The first byte of the header is 224 for a numeric literal and 225 for a character literal. The second byte of the header is the length of the literal, not including the header bytes.

Several C functions are available to provide various services for ccall routines.

`int dbcgetflags(void);`

Return the values of the four flags. The low-order bit of the return value is the value of the eos flag; the next bit is the value of the equal flag; the next bit is the value of the less flag; the next bit is the value of the over flag. In other words:

`(dbcgetflags( )&0x01)` is eos
`(dbcgetflags( )&0x02)` is equal
`(dbcgetflags( )&0x04)` is less
`(dbcgetflags( )&0x08)` is over

`void dbcseteos(int value);`

Set or clear the value of the eos flag. If value is zero, clear the eos flag. If value is non-zero, set the eos flag.

`void dbcsetequal(int value);`

Set or clear the value of the equal flag. If value is zero, clear the equal flag. If value is non-zero, set the equal flag.

`void dbcsetless(int value);`

Set or clear the value of the less flag. If value is zero, clear the less flag. If value is non-zero, set the less flag.

`void dbcsetover(int value);`

Set or clear the value of the over flag. If value is zero, clear the over flag. If value is non-zero, set the over flag.

`int dbcgetparm(unsigned char **list);`

Return the next parameter for the list of non-positional and keyword parameters. **list** points to an array of 6 pointers that point to the 6 possible parameters of an operand. The first pointer points to a keyword literal. The next 5 pointers point to the variables or literals that make up the parameters of the keyword operand. If the operand was a non-positional operand (it did not contain a keyword), the first pointer is null.

If **dbcgetparm** is called after the last parameter has been retrieved, -1 is returned. is called after the last parameter has been retrieved, -1 is returned. Otherwise the value returned is the number of parameters after the keyword. This value can be 0, 1, 2, 3, 4 or 5.

`void dbcresetparm(void);`

The position for **dbcgetparm** is reset so the next **dbcgetparm** retrieves the first non-positional or keyword operand in the statement.

void dbcdeath(int code);

Terminate the DB/C DX runtime with the internal error number specified by **code**.

The following C code ascertains what kind of variable is the first variable from the list and then calculates the size of the variable:

```
ccall(unsigned char *name, unsigned char **list) {
        int formptr; /* form pointer */
        int loglen; /* logical length */
        int maxlen; /* maximum length */
        unsigned char *ptr; /* work pointer */
        int i; /* work variable */
        ptr = list[0];
        if (ptr[0] < 0x80) { /* small CHAR var */
                formptr = ptr[0];
                loglen = ptr[1];
                maxlen = ptr[2];
        }
        else if ((ptr[0] & 0xe0) == 0x80) { /* FORM var */
                maxlen = ptr[0] & 0x7F;
        }
        else if ((ptr[0] & 0xFC) == 0xF8) { /* FLOAT var */
                maxlen = ((ptr[0] & 3) << 3) + (ptr[1] >> 5);
                if (i = (ptr[1] & 0x1F)) maxlen += i + 1;
        }
        else if (ptr[0] == 0xFC) { /* INT var */
                maxlen = ptr[1] & 0x1f;
        }
        else if (ptr[0] == 0xF0) { /* large CHAR var */
                formptr = ptr[1] + ((int) ptr[2] << 8);
                loglen = ptr[3] + ((int) ptr[4] << 8);
                maxlen = ptr[5] + ((int) ptr[6] << 8);
        }
        else if (ptr[0] == 0xE0) { /* numeric literal */
                maxlen = ptr[1];
        }
        else if (ptr[0] == 0xE1) { /* character literal */
                maxlen = ptr[1];
        }
        /* else unsupported type */
}
```

Writing Communication Routines

The communications statements (comopen, send, recv, etc.) are described in the Executable Statements chapter of this manual. If support for different hardware or other features is required, then the standard functions of the communications routines can be replaced with custom-written routines. This section describes how to write these routines.

The **dxreadme.txt** file found on the DB/C DX distribution media contains information describing how to compile and link a ccall function with the DB/C DX runtime. The communications module is incorporated with the DB/C DX runtime in the same manner. The compiled communications routines are contained in a module named **com.obj** or **com.o**, depending on the operating system.

All the functions described in the following paragraphs must be provided in the **com** module. All these functions except **comexit** may call the abnormal termination function named **dbcdeath**. In the following descriptions, true is 1 and false is 0.

```
int cominit(void);
```

Initialize the communications routines. This routine will be called once by the DB/C DX runtime before any other com routines are called. If **cominit** is successful, zero is returned; otherwise, a non-zero value is returned. If a non-zero value is returned, than an I/O error occurs.

```
void comexit(void);
```

Terminate all channels and clear the communications functions for program termination. This function will be called once at normal or abnormal (DB/C DX internal error) termination of the DB/C DX runtime.

```
int comopen(char *command, int *handle);
```

Open a communications channel as specified by command. A channel corresponds to a comfile declaration. The parameter **`command`** points to a zero delimited string that contains the file or device name, or TCP/IP socket description specified in the comopen statement. The parameter **`handle`** is a pointer to an integer. If **comopen** is successful, handle should be set to a positive channel number. If comopen is successful, zero is returned; otherwise, a non-zero value is returned. If a non-zero value is returned, than an I/O error occurs.

```
int comclose(int channel);
```

Close the communications channel specified by the parameter **channel**, which is the channel number set by **comopen**. If **comclose** is successful, zero is returned; otherwise, a non-zero value is returned. If a non-zero value is returned, than an I/O error occurs.

```
int comrecv(int channel, int evtid, int timeout, int count);
```

Initiate the receiving of a message through the **channel** specified by the parameter channel. This function is called in response to the recv statement. The parameter **evtid** is the event handle that should be set by **evtfunc1** when the receive is completed. The function should clear the event handle by calling **evtfunc2** before beginning the receive. The parameter **timeout** is the amount of time that the com module should wait before considering the receive attempt to have failed. A positive value designates a positive number of seconds. A value of zero designates an immediate timeout. A negative value designates that no timeout should occur. The parameter count is the maximum number of bytes that should be received by this call. If **comrecv** successfully initiates the receive process, the status of the channel is set to receive pending (**COM_RECV_PEND**) and **comrecv** returns zero; otherwise, a non-zero value is returned. If a non-zero value is returned, than an I/O error occurs.

```
int comrecvget(int channel, unsigned char *msgptr, int *msglen);
```

Return the message received by **`comrecv`**. This function is called in response to the event handle provided by **comrecv** being set. The parameter **msgptr** is a pointer to the area in which the message will be stored. The parameter **msglen** is a pointer to an integer that is the maximum length for the message to be stored in the message area pointed to by **msgptr**. The **msgptr** area will remain valid until a **comclear** call is made by the DB/C DX runtime. On return, the integer pointed to by **msglen** will contain the actual length of the message that was received. If **comrecvget** is successful, than zero is returned; otherwise, a non-zero value is returned. If a non-zero value is returned, than an I/O error occurs.

```
int comsend(int channel, int evtid, int timeout, unsigned char *msgptr,
         unsigned char *msglen);
```

Initiate the sending of a message through the channel specified by the parameter **channel**. This function is called in response to the send statement. The parameter **evtid** is the event handle that should be set by **evtfunc1** when the send is completed. The function should clear the event handle by calling **evtfunc2** before beginning the send. The parameter **timeout** is the amount of time that the com module should wait before considering the send attempt to have failed. A positive value designates a positive number of seconds. A value of zero designates an immediate timeout. A negative value designates that no timeout should occur. The parameter **msgptr** is a pointer to the message to be sent. The parameter **msglen** contains the length of the message. The **msgptr** area will remain valid until a **comclear** call is made by the DB/C DX runtime. If **comsend** successfully initiates the send process, the status of the channel is set to send pending (**COM_SEND_PEND**) and **comsend** returns zero; otherwise, a non-zero value is returned. If a non-zero value is returned, than an I/O error occurs.

```
int comstat(int channel, int *status);
```

Provide the status for the channel specified by the parameter **channel**. The parameter **status** is a pointer to an integer that will be set to the status of the channel. If **comstat** is successful, zero is returned; otherwise, a non-zero value is returned. If a non-zero value is returned, than an I/O error occurs. The bit values for the status are:

COM_SEND_PEND =	**0x00000001**	comsend in progress
COM_SEND_DONE =	**0x00000002**	comsend completed successfully
COM_SEND_TIME =	**0x00000004**	comsend timed out
COM_SEND_ERROR =	**0x00000008**	comsend ended with an error
COM_RECV_PEND =	**0x00000010**	comrecv in progress
COM_RECV_DONE =	**0x00000020**	comrecv completed successfully
COM_RECV_TIME =	**0x00000004**	comrecv timed out
COM_RECV_ERROR =	**0x00000080**	comrecv ended with an error
COM_PER_ERROR =	**0X00010000**	permanent error condition exists

```
int comclear(int channel, int *status);
```

Clear the pending operations for the channel specified by the parameter **channel**, and set the status to idle. The parameter **status** is a pointer to an integer that is set to the status of the channel before it is cleared. If a permanent error condition exists, it should not be cleared. On success, **comclear** returns zero; otherwise a nonzero value is returned. If a non-zero value is returned, than an I/O error occurs.

```
int comsclear(int channel, int *status);
```

Clear the send operation for the channel specified by the parameter **channel**. Set the integer pointed to by the parameter **status** to the status of the channel before it is cleared. On success, **comsclear** returns zero; otherwise a non-zero value is returned. If a non-zero value is returned, than an I/O error occurs.

```
int comrclear(int channel, int *status);
```

Clear the receive operation for the channel specified by the parameter **channel**. Set the integer pointed to by the parameter **status** to the status of the channel before it is cleared. On success, **comrclear** returns zero; otherwise a non-zero value is returned. If a non-zero value is returned, than an I/O error occurs.

```
int comctl(int channel, unsigned char *msgin, int msginlen,
         unsigned char *msgout, int *msgoutlen);
```

Send and receive control information for the channel specified by the parameter **channel**. . The parameter **msgin** points to a zero delimited string that contains the incoming control information. The length of this control information is specified by the parameter **msginlen**. **comctl** returns control information in a zero delimited string in the area pointed to by **msgout**. The maximum length of this string is specified in the integer pointed to by **msgoutlen**. On return, **msgoutlen** should be set to the length of the string placed in **msgout**. On success, **comctl** returns zero; otherwise a non-zero value is returned. If a non-zero value is returned, than an I/O error occurs.

Implementing Native Files

Native files are files that are accessed through file or ifile declarations which have the native option specified. This section describes how to write the functions that support this feature.

The **dxreadme.txt** file distributed with DB/C DX contains information describing how to compile and link a ccall function with the DB/C DX runtime. The native file interface module is incorporated with the runtime in the same manner. The compiled routines are contained in a module named **nio.obj** or **nio.o**, depending on the operating system.

All the functions described in the following paragraphs must be provided in the nio module. All these routines may call the DB/C DX abnormal termination routine named **dbcdeath**.

For all nio functions, a negative return value indicates that the operation was unsuccessful and an error may have occurred. A return value of -1 indicates that a system error was encountered or the called interface is not available. A return value of -98 is reserved for user-defined errors. A return value of -99 is reserved for DB/C DX defined errors.

```
int nioopen(char *name, int index, int exclusive);
```

Open a file. The parameter **name** points to a zero delimited string that contains the file name specified in the open statement. If the **index** parameter is zero, then the open is from a file declaration. Otherwise, the open is from an ifile declaration. The value of **index** is the value of keylen or -1 if keylen is not specified. The value of **index** may be used to refer to the index key in a file that contains multiple keys or it may be used for any user-defined purpose. If the parameter **exclusive** is non-zero, the file should be opened in exclusive mode. Otherwise, open the file in shared mode. If the return value is non-negative, then the open succeeded and the return value is the handle to be used for further access. If the return value is negative, then a DB/C DX runtime I/O error occurred. A return value of -2 indicates that the file was not found. A return value of -3 indicates that an access violation occurred when the file was opened.

```
int nioprep(char *name, char *options, int index);
```

Prepare a file. The parameter **name** points to a zero delimited string that contains the file name specified in the open statement. The parameter **options** points to a zero delimited string that contains prepare options. If the **index** parameter is zero, then the open is from a file declaration. Otherwise, the open is from an ifile declaration. The value of **index** is the value of keylen or -1 if keylen is not specified. The value of **index** may be used to refer to the index key in a file that contains multiple keys or it may be used for any user-defined purpose. If the return value is non-negative, then the prepare succeeded and the return value is the handle to be used for further access. If the return value is negative, then a DB/C DX runtime I/O error occurred. A return value of -2 indicates that the directory or drive was not found. A return value of -3 indicates that an access violation occurred when the file was created.

```
int nioclose(int handle);
```

Close the file specified by the parameter **handle**. If the return value is non-negative, then the operation was successful. If the return value is negative, then a DB/C DX runtime I/O error occurred.

```
int niordseq(int handle, unsigned char *buffer, int length);
```

Read the next record sequentially. The parameter **handle** specifies the file. The record is moved to the area pointed to by the parameter **buffer**. The record length is specified by the parameter **length**. If the return value is non-negative, then the operation was successful and the return value is the length of the record that was moved to the buffer. If the end of file is encountered, the return value is -2. Otherwise, if the return value is negative, then a DB/C DX runtime I/O error occurred.

```
int niowtseq(int handle, unsigned char *buffer, int length);
```

Write a record sequentially. The parameter **handle** specifies the file. The record is in the area pointed to by the parameter **buffer**. The record length is specified by the parameter **length**. If the return value is non-negative, then the operation was successful. If the return value is negative, then a DB/C DX runtime I/O error occurred.

`int niordrec(int handle, long recnum, unsigned char *buffer, int length);`

Read a record by record number. The parameter **handle** specifies the file. The parameter **recnum** contains the record number (zero is the first record). The record is moved to the area pointed to by the parameter **buffer**. The record length is specified by the parameter length. If the return value is non-negative, then the operation was successful and the return value is the length of the record that was moved to the buffer. If the end of file is en countered, the return value is -2. Otherwise, if the return value is negative, then a DB/C DX runtime I/O error occurred.

`int niowtrec(int handle, long recnum, unsigned char *buffer, int length);`

Write a record by record number. The parameter **handle** specifies the file. The parameter **recnum** contains the record number (zero is the first record). The record is in the area pointed to by the parameter **buffer**. The record length is specified by the parameter **length**. If the return value is non-negative, then the operation was successful. If the return value is negative, then a DB/C DX runtime I/O error occurred.

```
int niordkey(int handle, unsigned char *key, unsigned char *buffer,
         int length);
```

Read a record by key. The parameter **handle** specifies the file. The parameter **key** points to a zero terminated string that specifies the lookup key. The record is moved to the area pointed to by the parameter **buffer**. The record length is specified by the parameter **length**. If the return value is non-negative, then the operation was successful and the return value is the length of the record that was moved to the buffer. If the key could not be found, the return value is -2. Otherwise, if the return value is negative, then a DB/C DX runtime I/O error occurred.

`int niordks(int handle, unsigned char *buffer, int length);`

Read the record associated with the next key. The parameter **handle** specifies the file. The record is moved to the area pointed to by the parameter **buffer**. The record length is specified by the parameter **length**. If the return value is non-negative, then the operation was successful and the return value is the length of the record that was moved to the buffer. If the end of file is encountered, the return value is -2. Otherwise, if the return value is negative, then a DB/C DX runtime I/O error occurred.

`int niordkp(int handle, unsigned char *buffer, int length);`

Read the record associated with the previous key. The parameter **handle** specifies the file. The record is moved to the area pointed to by the parameter **buffer**. The record length is specified by the parameter **length**. If the return value is non-negative, then the operation was successful and the return value is the length of the record that was moved to the buffer. If the beginning of file is encountered, the return value is -2. Otherwise, if the return value is negative, then a DB/C DX runtime I/O error occurred.

`int niowtkey(int handle, unsigned char *buffer, int length);`

Write the record and key. The parameter **handle** specifies the file. The record is in the area pointed to by the parameter **buffer**. The record length is specified by the parameter **length**. If the return value is non-negative, then the operation was successful. If the key is a duplicate and duplicates are not allowed, the return value is -2. Otherwise, if the return value is negative, then a DB/C DX runtime I/O error occurred.

`int nioupd(int handle, unsigned char *buffer, int length);`

Update the current record. The parameter **handle** specifies the file. The record is in the area pointed to by the parameter **buffer**. The record length is specified by the parameter **length**. If the return value is non-negative, then the operation was successful. If the return value is negative, then a DB/C DX runtime I/O error occurred.

`int niodel(int handle, unsigned char *key);`

Delete a record and key. The parameter **handle** specifies the file. The parameter **key** points to a zero delimited string that is the key of the record to be deleted. If the return value is non-negative, then the operation was successful. If the key was not found, the return value is -2. Otherwise, if the return value is negative, then a DB/C DX runtime I/O error occurred. Depending on the system, all other keys associated with this record may or may not be deleted by this routine.

`int niolck(int *handle);`

Lock one or more files. This function is called when a filepi statement is executed for one or more native files. The parameter **handle** is the pointer to an array of file handles. The last array element will be zero. If the return value is zero, then the operation was successful. If the return value is negative, then a DB/C DX runtime I/O error occurred.

`int nioulck(void);`

Release the locks issued by niolck. This function is called when a filepi 0 statement is executed or when a filepi expires. If the return value is zero, then the operation was successful. If the return value is negative, then a DB/C DX runtime I/O error occurred.

`int nioclru(void);`

Close the least recently used file. This function is called by the DB/C DX runtime or the nio routines when there are no more available handles for the process or from the operating system. If the return value is zero, then the operation was successful and a file was closed. If the return value is -1, then the operation was unable to close a file.

`int nioerror(void);`

Return the error that is specific to the operating system. This function may be called after another nio function returns an error code.

Writing Device Support Routines

The device variable declaration defines a device. open, close, query, change, load, store, link, and unlink are operations that are allowed to act on a device variable. This section describes how to write the functions to support these operations.

The **dxreadme.txt** file distributed with DB/C DX contains information describing how to compile and link a ccall function with the DB/C DX runtime. The device support routines module is incorporated with the DB/C DX runtime in the same manner. The compiled functions are contained in a module named **dsr.obj** or **dsr.o**, depending on the operating system.

All functions described in the following paragraphs must be provided in the dsr module. All the functions may call the abnormal termination function named **dbcdeath**.

`int dsropen(char *name);`

Open a device. The parameter **name** points to a zero delimited string that contains the name of the device specified in the open statement. A return value of -1 indicates that the device was not found. A return value of -2 indicates that a system error was encountered. If the return value is negative, then a DB/C DX runtime I/O error occurred. If the return value is non-negative, then the open was successful and the return value is the handle to be used for further access.

`int dsrclose(int handle);`

Close the device specified by **handle**. If **dsrclose** is successful, it returns true, otherwise it returns false. If false is returned, an I/O error occurs.

`int dsrquery(int handle, char *function, char *buffer, int *length);`

Perform the query operation. The parameter **handle** specifies the device. The parameter **function** is a pointer to a zero terminated string that is the logical string of the second operand in the query operation. The parameter **buffer** is a pointer to an area that contains the information from the list of variables in the query operation. **length** is a pointer to the size of buffer. The **dsrquery** function may change the contents or size of the buffer area. Those changes will be reflected in the variables in the list. If **dsrquery** is successful, it returns true. If it is unsuccessful, it returns false and an I/O error occurs.

`int dsrchange(int handle, char *function, char *buffer, int length);`

Perform the change operation. The parameter **handle** specifies the device. The parameter **function** is a pointer to a zero terminated string that is the logical string of the second operand in the change operation. The parameter **buffer** is a pointer to an area that contains the information from the list of variables in the change operation, query operand. **length** is the size of the buffer. If **dsrchange** is successful, it returns true. If it is unsuccessful, it returns false and an I/O error occurs.

`int dsrload(int handle, struct imgvar **imagevar);`

Load an image from a device. The parameter **handle** specifies the device. The parameter **imagevar** points to a pointer to a structure that describes information associated with the image variable specified in the load statement. If **dsrload** is successful, it returns true. If it is unsuccessful, it returns false and an I/O error occurs.

`int dsrstore(int handle, struct imgvar **imagevar);`

Store an image onto a device. The parameter **handle** specifies the device. The parameter **imagevar** points to a pointer to a structure that is the same as for **dsrload**. If **dsrstore** is successful, it returns true. If it is unsuccessful, it returns false and an I/O error occurs.

`int dsrlink(int handle, struct queue ** qhandle);`

Link a device with a queue. The parameter **handle** specifies the device. The parameter **qhandle** specifies the queue. If **dsrlink** is successful, it returns true. If it is unsuccessful, it returns false and an I/O error occurs.

`int dsrunlink(int handle, struct queue ** qhandle);`

Unlink a device from a queue. The parameter **handle** specifies the device. The parameter **qhandle** specifies the queue. If **dsrunlink** is successful, it returns true. If it is unsuccessful, it returns false and an I/O error occurs.

The dsr routines may use these three queue handling functions:

`int queget(struct queue ** qhandle, char *message, int length);`

Obtain a message from a queue. The parameter **qhandle** specifies the queue. The parameter **message** points to the area to which the message is moved. The parameter **length** is an integer that contains the size of the area pointed to by message. If no message is available, **queget** returns zero. If a message is moved, **queget** returns the length of the message. If an error occurs, -1 is returned.

`int queput(struct queue ** qhandle, char *message, int length);`

Place a message into a queue. The parameter **qhandle** specifies the queue. The parameter **message** points to the area that contains the message. **length** is the length of that message. If **queput** is successful, it returns zero. Otherwise it returns non-zero.

`int quewait(struct queue ** qhandle);`

Wait for a message to become available in a queue. **qhandle** specifies the queue. **quewait** returns true if successful. Otherwise, it returns false.

File Formats

Standard Text Files

Standard text files (also known as DB/C type files) are portable files. They consist of control characters and the 95 ASCII printable characters. Standard text files are the default type of data file created by DB/C DX utilities and the DB/C DX runtime.

Each record is from 0 to 65500 characters long.

The end of the record is indicated by one byte with value 0xFA. Deleted records are replaced by bytes with value 0xFF, including the end of record character. The end of the file is indicated by one byte with value 0xFB.

The characters that make up a record consist of the 95 printable ASCII characters plus compression bytes with values 0x80 through 0xF9. The values 0x80 through 0xF8 correspond with a two character to one byte compression scheme, as shown in this table:

`0x80` is `"00"`
`0x81` is `"01"`
`0x82` is `"02"`
`0x83` is `"03"`
`0x84` is `"04"`
`0x85` is `"05"`
`0x86` is `"06"`
`0x87` is `"07"`
`0x88` is `"08"`
`0x89` is `"09"`
`0x8A` is `"0."`
`0x8B` is `"10"`
`0x8C` is `"11"`
`0x8D` is `"12"`
`0x8E` is `"13"`
`0x8F` is `"14"`
`0x90` is `"15"`
`0x91` is `"16"`
`0x92` is `"17"`
`0x93` is `"18"`
`0x94` is `"19"`
`0x95` is `"1."`
`0x96` is `"20"`
`0x97` is `"21"`
`0x98` is `"22"`
`0x99` is `"23"`
`0x9A` is `"24"`
`0x9B` is `"25"`
`0x9C` is `"26"`
`0x9D` is `"27"`
`0x9E` is `"28"`
`0x9F` is `"29"`
`0xA0` is `"2."`
`0xA1` is `"30"`
`0xA2` is `"31"`
`0xA3` is `"32"`
`0xA4` is `"33"`
`0xA5` is `"34"`
`0xA6` is `"35"`
`0xA7` is `"36"`
`0xA8` is `"37"`
`0xA9` is `"38"`
`0xAA` is `"39"`
`0xAB` is `"3."`
`0xAC` is `"40"`
`0xAD` is `"41"`
`0xAE` is `"42"`
`0xAF` is `"43"`
`0xB0` is `"44"`
`0xB1` is `"45"`
`0xB2` is `"46"`
`0xB3` is `"47"`
`0xB4` is `"48"`
`0xB5` is `"49"`
`0xB6` is `"4."`
`0xB7` is `"50"`
`0xB8` is `"51"`
`0xB9` is `"52"`
`0xBA` is `"53"`
`0xBB` is `"54"`
`0xBC` is `"55"`
`0xBD` is `"56"`
`0xBE` is `"57"`
`0xBF` is `"58"`
`0xC0` is `"59"`
`0xC1` is `"5."`
`0xC2` is `"60"`
`0xC3` is `"61"`
`0xC4` is `"62"`
`0xC5` is `"63"`
`0xC6` is `"64"`
`0xC7` is `"65"`
`0xC8` is `"66"`
`0xC9` is `"67"`
`0xCA` is `"68"`
`0xCB` is `"69"`
`0xCC` is `"6."`
`0xCD` is `"70"`
`0xCE` is `"71"`
`0xCF` is `"72"`
`0xD0` is `"73"`
`0xD1` is `"74"`
`0xD2` is `"75"`
`0xD3` is `"76"`
`0xD4` is `"77"`
`0xD5` is `"78"`
`0xD6` is `"79"`
`0xD7` is `"7."`
`0xD8` is `"80"`
`0xD9` is `"81"`
`0xDA` is `"82"`
`0xDB` is `"83"`
`0xDC` is `"84"`
`0xDD` is `"85"`
`0xDE` is `"86"`
`0xDF` is `"87"`
`0xE0` is `"88"`
`0xE1` is `"89"`
`0xE2` is `"8."`
`0xE3` is `"90"`
`0xE4` is `"91"`
`0xE5` is `"92"`
`0xE6` is `"93"`
`0xE7` is `"94"`
`0xE8` is `"95"`
`0xE9` is `"96"`
`0xEA` is `"97"`
`0xEB` is `"98"`
`0xEC` is `"99"`
`0xED` is `"9."`
`0xEE` is `".0"`
`0xEF` is `".1"`
`0xF0` is `".2"`
`0xF1` is `".3"`
`0xF2` is `".4"`
`0xF3` is `".5"`
`0xF4` is `".6"`
`0xF5` is `".7"`
`0xF6` is `".8"`
`0xF7` is `".9"`
`0xF8` is `".."`

The byte value 0xF9 is a blank compression leader character. The one byte following it is the number of blank characters that are represented by these two bytes. The allowable values for the byte following the 0xF9 are 3 through 248.

Data Text Files

Data text files are portable ASCII type text files. Data text files are created with the -d option on the utilities and records. Each record is 0 to 65500 characters long. The characters in a record are the 95 printable ASCII characters. The end of the record is indicated by one byte with value 0x0A (ASCII LF). Deleted records are replaced by bytes with value 0x7F (ASCII DEL), including the end of record character. The length of the file indicates the end of file, thus there is no end of file byte.

Native Text Files

Native text files are non-portable type text files that are correct for the operating system on which they are created. Native text files are created with the -t option on the utilities and with the text keyword on file, ifile and afile statements in DB/C programs. The maximum size of a native text file record is 65500 characters.

Index Files

Index files consist of a header block, branch blocks, leaf blocks, delete blocks and record reclamation blocks. The default index block size is 1024. Valid index block sizes are 512, 1024, 2048, 4096, 8192, and 16384.

In the following tables, HHMMLL means a six byte binary field with the high order byte at the lowest file offset and the low order byte at the highest file offset. The high order bit of the high order byte of an HHMMLL field is always 0.

All character encoding is ASCII.

The header block is located at the beginning of the index file. The format of the header block is:

Offset	**Length**	**Description**
0	1	`"L"`
1	6	index file offset of first block in linked list of deleted index file blocks or 0 if none (HHMMLL)
7	6	index file offset of first block in linked list of space reclamation blocks or 0 if none (HHMMLL)
13	6	index file offset of top branch or leaf block in index tree or 0 if no records in index (HHMMLL)
19	6	index file offset of highest used block (HHMMLL)
25	16	blanks
41	5	index block size (ASCII digits blank filled on left)
46	8	blanks
54	1	`"L"` if index is using leading character compression
55	1	`"T"` if **-t** is specified on index command line, blank if **-t** is not specified
56	1	`"D"` if **-d** is specified on index command line, blank if -d is not specified
57	1	`"S"` if space reclamation is supported, blank if not supported
58	2	key size. If less than 10 then blank followed by one ASCII digit value **1**-**9**. Else if less than 100 then two ASCII digits. Else if less than 255 then first character is **A** through **P** and second character is ASCII digit **0**-**9**.
60	4	blanks

continued on next page

Offset	**Length**	**Description**
64	5	record length if space reclamation supported (ASCII digits blank filled on left)
69	30	blanks
98	2	`"10"` index file version number
100	1	0xFA
101	variable	text file name
next	1	0xFA
next	variable	command line parameters, each separated with one byte value 0xFA and terminated with two consecutive 0xFA bytes
next	variable	zero or more bytes of 0xFF through end of block

The format of a branch block is:

Offset	**Length**	**Description**
0	1	`"U"`
1	6	index file offset of lower level branch or leaf block that contains keys less than next key in this branch block (HHMMLL)
7	*klen*	key value of a text file record (*klen* is key length)
klen + 7	6	text file offset of record associated with previous key (HHMMLL)
klen + 13	6	index file offset of lower level branch or leaf block that contains keys greater than previous key and less than next key (HHMMLL)
		note: the previous three fields may be repeated as many times as will fit into one block
next	variable	zero or more bytes of 0xFF through end of block

The format of a leaf block is:

Offset	**Length**	**Description**
0	1	`"V"`
1	*klen*	key value of a text file record (*klen* is key length)
klen + 1	6	text file offset of record associated with previous key (HHMMLL)
		note: the following three fields are may not exist or may be repeated as many times as will fit into one block
next	1	one byte binary value (*n*) is the number of characters omitted from the left side of the key value in following field because they duplicate characters from the left side of the previous key in this leaf block
next	*klen* - *n*	rightmost characters of key value of text file record that are different from previous key in this leaf block
next	6	text file offset of record associated with previous key (HHMMLL)
next	variable	zero or more bytes of 0xFF through end of block

The format of a deleted block is:

Offset	Length	Description
0	1	`"D"`
1	6	index file offset of next deleted block (HHMMLL)
7	variable	multiple bytes of 0xFF through end of block

The format of the space reclamation block is:

Offset	Length	Description
0	1	`"R"`
1	6	file offset of deleted record (HHMMLL)
next	variable	multiple bytes of 0xFF through end of block

AIM Index Files

Associate index method index files consist of a header block and one or more extent blocks. The header block size is 1024.

In the following table, HHMMLL means a six byte binary field with the high order byte at the lowest file offset and the low order byte at the highest file offset. The high order bit of the high order byte of an HHMMLL field is always zero.

All character encoding is ASCII.

The header block is located at the beginning of the AIM index file. The format of the header block is:

Offset	Length	Description
0	1	`"A"`
1	6	AIM index file offset of the next extent or 0 if none (HHMMLL)
7	6	the record number + 1 of a record that has been deleted and is found in this extent or 0 if none (HHMMLL)
13	6	the number of records in each slot in this extent (HHMMLL)
19	13	blanks
32	5	number of slots (Z value) (ASCII digits blank filled on left)
36	4	blanks
41	5	data file record length (ASCII digits blank filled on left)
46	11	blanks
57	1	`"Y"` if upper and lower case are distinct, `"N"` if not distinct
58	1	match character
59	1	`"V"` if variable length records in text file if variable length records in text file,`"F"` if fixed length records in text file, `"S"` if fixed length records and space reclamation

60	6	number of records allocated to allocate in each secondary extent (HHMMLL)
65	34	blanks
98	2	`"10"` AIM index file version number
100	1	0xFA
101	variable	text file name
next	1	0xFA
next	variable	variable command line parameters, each separated with one byte value 0xFA and terminated with two consecutive 0xFA bytes
next	variable	zero or more bytes of 0xFF through end of block

For AIM files, the rest of file consists of one or more extents. The first extent (or primary extent) always exists. Each extent begins with a preface. The primary extent preface is just the 1024-byte header block. Secondary extents begin with a preface of 28 bytes. Their format is the same as the first 28 bytes of the AIM header block. After the preface, each extent contains information broken into slots. There are (Z value) slots in each extent. The first slot begins at file offset 1024 for the primary extent and at the first byte after the 28-byte preface in secondary extents. The number of records in each extent is always evenly divisible by eight. Each slot consists of (N / 8) bytes where N is the number of records in this extent. Each bit in the slot corresponds to a record. If a record contains a pattern that maps to this slot number, then the bit for that record will be on; otherwise it will be off. For variable length records, the record number is ((FPOS - 1) / 256) where FPOS is the file position of the first byte of the record.

The AIM pattern coding mechanism uses a hashing algorithm. The five coding patterns are:

Leftmost character (L1)	<left byte *fn*>, 31, <left byte>
Two leftmost characters (L2)	<two left bytes *fn*>, <left byte>, <left + one byte>
Rightmost character (R1)	<right byte *fn*>, 30, <right byte>
Two rightmost characters (R2)	<two right bytes *fn*>, <right byte>, <right + one byte>
Three characters (F)	<three byte float> <byte 1>, <byte 2>, <byte 3>

fn stands for field number. It is the key field number (first one is zero). Each of the three bytes is anded with hex 1F (same as modulo 32). The resulting five bits from each byte are appended to create an unsigned 15-bit number. The decimal range of a 15-bit number is 0 to 32767. This number is then taken modulo the Z value (number of slots), to arrive at the slot number to which the coding pattern hashes. The actual AIM lookup method will check the actual record to see if the desired lookup key actually exists in the record. For each key field specified in the aimdex command that is not an excluded field, the following patterns are used:

key field length 1	L1
key field length 2	L1, R1
key field length 3	L1, R1, L2, R2
key field length > 3	L1, R1, L2, R2 and zero or more F

Float fields are only hashed into a slot for each group of three adjacent non-blank characters.

DB/C Programming Language General Information

Statement Structure

All statements follow this general format:

label operation operands comments

label	is a string of characters. A label may contain an upper-case character (**A**-**Z**), a lower-case character (**a**-**z**), a dollar sign (**$**), a digit (**0**-**9**), a period (**.**), an underscore (**_**), or an “at” sign (**@**). The first character in the string must be an upper-case character, a lower-case character, or a dollar sign. The maximum length is 31 characters. The label is optional in some statements. If the first character of a program line is a blank or a tab, the line does not contain a label. If a label exists, one or more blank or tab characters must separate the label and the operation. There are two types of labels: the data label and the execution label. A data label may have the same name as an execution label. However, a data label cannot have the same name as another data label, and an execution label cannot have the same name as another execution label.
operation	(also called the verb), is a string of characters. An operation may contain an upper-case character (**A**-**Z**), a lower-case character (**a**-**z**), a dollar sign (**$**), a digit (**0**-**9**), a period (**.**), an underscore (**_**), or an “at” sign (**@**). The first character in the string must be an upper-case character or a lowercase character. The maximum length is 31 characters. No distinction is made between lower-case and upper-case characters. One or more blank or tab characters separate the operation from the operand.
operands	may be required by the operation. See the description of each operation for the format of its operands. Operands may be labels, literals, constants, variables, or ***** control directives. (These terms are defined later in this chapter.) Operands are separated by commas, semicolons, colons, and prepositions. The valid prepositions are: **by**, **to**, **of**, **from**, **using**, **with**, **in**, and **into**. The separators used with particular operands are defined in the section describing the operation. Some operations require a list of operands. A comma separates each operand in a list. If a colon is placed after the last operand of the current line, the list continues on the next line. The continuation line must start with one or more blank or tab characters.
comments	may follow the operands. One or more blank or tab characters separate the comments from the operands. Some operations allow optional operands. With these operations, it can be ambiguous whether certain text is intended to be an operand or a comment. To eliminate this ambiguity, a period can be used as a comment delimiter. The period is used to indicate the start of a comment. The remainder of the line is considered to be a comment. The maximum line length is 255 characters.

Comment Lines

A line that has **+** or ***** or **.** as the first character is a comment statement and is ignored by the compiler. The compiler also ignores a completely blank line.

A comment line may appear anywhere in a program.

A comment line may be embedded between two lines that constitute one statement that are logically connected by means of a colon.

Standard Character Set

The standard character set for DB/C is:

```
blank
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789
!"#$%&'()*+,-./:;<=>?[ ]^_@{\|}~
```

The current implementation of DB/C DX is in ASCII, but DB/C DX is designed to work with other character encoding sequences and different standard character sets.

Character Literals

A character literal is a string of characters from the standard character set that begins and ends with quotation marks (`"`). The maximum length of a literal is 253 characters.

There are two ways to include a quotation mark character in a literal. The first way is to specify two consecutive quotation marks (`""`). The second way is to use the pound sign (`#`) forcing character. The character immediately to the right of the pound sign is included in the literal character string, regardless of what it is. The forcing character itself is removed from the literal. For example, to include the pound sign character in a literal, use the sequence `##`.

A special case of the literal string is the one-character literal used by the equate, cmove, cmatch, and trap statements. The forcing character is not used in the one-character literal used in these operations.

Numeric Literals

A numeric literal consists of a valid DB/C number. A DB/C number begins with zero or more blanks, optionally followed by a minus sign, followed by zero or more digits, optionally followed by a decimal point. If a decimal point is included, it must be followed by one or more digits. At least one digit must be in the string. No other characters are allowed in the string. In a valid DB/C number, the total number of digits cannot exceed 30.

Constants

In this manual, the word "constants" refers to numeric literals, character literals, numeric constants, integer constants, hexadecimal constants, or octal constants.

A numeric constant is a string of digits, with or without a decimal point, that makes up a number. In certain cases, zero and negative values are also allowed.

An integer constant is a string of one or more digits that makes up an integral value. In certain cases, zero and negative values are also allowed. A decimal point is not allowed.

A hexadecimal constant is a string of digits and letters that represents a one-byte hexadecimal value. A hexadecimal constant starts with 0X or 0x and contains digits and the letters A-F or a-f.

An octal constant is a string of digits from 0-7 that starts with a 0.

Character and Numeric Variables

A character variable contains a string of one or more characters. There are two numeric indexes associated with each character variable: the form pointer and the length pointer. The value of the form pointer is always less than or equal to the value of the length pointer. The value of the length pointer is always less than or equal to the maximum size of the variable.

The logical string of a character variable is those characters from the form pointer to the length pointer, inclusive. If the form pointer is zero, the logical string is null, regardless of the value of the length pointer. The logical length of a character variable is the number of characters in the logical string.

There are three types of numeric variables: decimal, integer, and float.

A decimal numeric variable contains up to 30 digits plus a decimal point. The internal representation is decimal characters, so all calculations are exact except division which is exact to a power of ten.

An integer numeric variable contains positive and negative integral values. The internal representation is a signed 32 bit 2's complement value, so the minimum value is -2147483648 and the maximum value is 2147483647.

A float numeric variable contains floating point values. The internal representation is an eight-byte float value appropriate for the hardware.

The NULL Value

Character and numeric variables may contain a special value called the NULL value. The logical string of a variable that has the NULL value is a zero length string. The setnull statement sets the value of a variable to NULL. When a variable is the destination of any statement or operation except the setnull statement, its value is set to "not NULL."

Flags

There are four flags used in DB/C programs: eos, equal, less, and over. eos stands for end-of-string. equal is syntactically equivalent to zero. over stands for overflow.

Flags have two states: set (or true) and clear (or false). Most operations affect one or more flags. Some operations are affected by the state of one or more flags.

The pseudo-flag greater is used by some operations. It is considered to be set if both equal and less are clear. Otherwise, it is considered to be clear.

Expressions and Operators

An expression is a series of variables, constants, parentheses, and operators. Expressions may be used in place of constants in most operations. Logical expressions are used in if, while, and until statements.

Expressions must always be enclosed in parentheses. For example:

```
        compare ((j + 1) * k) to num1
```

If a colon is found in an expression, the expression continues on the next line.

The keywords **eos**, **equal**, **less**, **over**, and **greater** are treated as flags in expressions only if they are not defined else where as variables. For evaluation of the expression, the value of a flag is treated as an integer variable with the value 0 if the flag is cleared and with the value 1 if the flag is set.

Expressions evaluate to either a numeric value or to a character value. During evaluation of a character value, intermediate results and the final result are limited to a maximum of 256 characters. If truncation occurs during the evaluation of a character expression, then an E566 error occurs.

The flags are not affected by the evaluation of an expression.

There are two types of operators: unary and binary. Unary operators precede their operand. Binary operators are placed between their operands. Operands may be numeric or character variables or constants.

Unary Operators

+ and **–** are the unary positive and negative operators. Their operand must be numeric. **+** has no effect. **–** has the same effect as multiplying the operand by negative one. The resulting value is the same numeric type as the operand. If the result would be truncated, the size of the result is one greater than the operand (the left-size is increased to accommodate the minus sign). Otherwise the size of the result is the same as the size of the source.

int, **form**, and **float** are the integer, form, and float coercion operators. The resulting value is the specified numeric type. If the operand is a character variable or literal that does not contain a valid number, the resulting value is zero.

`char` is the character type coercion operator. The operand must be a numeric. The resulting value is a character string.

isnull is the NULL value test operator. The operand may be a numeric or a character variable. The result is a numeric value. If the operand has the NULL value, the result is one; otherwise, the result is zero.

sin, **cos**, **tan**, **arcsin**, **arccos**, **arctan**, **exp**, **log**, **log10**, **abs**, and **sqrt** are the sine, cosine, tangent, arcsine, arccosine, arctangent, exponential, natural logarithm, base 10 logarithm, absolute value, and square root operators, respectively. The operand must be numeric. The resulting value is float numeric type. The size is the same as the operand.

not is the logical negation operator. The operand must be numeric. **not** returns an integer value equal to zero if the value of the operand is non-zero. The result is one if the value of the operand is zero; otherwise the result is zero. The result is integer numeric type and the size is one.

length is the maximum length operator. The result is the length of the variable. The length of a numeric variable may be changed by the nformat statement. The length of a character variable is its maximum length, which may be changed by the sformat statement. The result is integer numeric type. If the operand is numeric, the size of the result is two. If the operand is character, the size of the result is 5.

size is the logical size operator. If the operand is numeric, the result of the **size** operator is a fractional value that is the *left-side***.***right-side* format of the numeric operand. The numeric type is decimal and the size is 2.1. If the *right-side* value would be greater than nine (because the operand has more than nine digits after the decimal point), then *left-side***.9** is the result of the size operator. If the operand is character, the result is an integer that is the logical length of the operand. If the form pointer of the character variable is zero, the logical length is zero. Otherwise, the logical length is equal to the length pointer minus the form pointer plus one. The result is integer numeric type and the size is 5.

fchar and **lchar** are the form pointed (or first) character and length pointed (or last) character operators. The operand must be a character variable. If the form pointer is not zero, the result is a character string of length one containing the specified character. Otherwise, the result is a character string of zero length.

squeeze, **chop** and **trim** are the squeeze, chop and trim operators. The operand must be a character variable. The result of the squeeze operation is a character string that contains all the non-blank characters from the operand. The result of the chop operation is a character string that contains all the characters in the operand except trailing blanks. The result of the trim operation is a character string that contains all the characters in the operand except the leading and trailing blanks.

formptr and **lengthptr** are the form pointer value and length pointer value operators. The operand must be a character variable. The result is an integer value that is the form pointer or the length pointer of the character variable. The result is integer numeric type and the size is 5.

Binary Operators

`+`, `-`, `*`, `/`, `%`, and `**` are the add, subtract, multiply, divide, modulus, and power operators. Their operands are numeric. For the power operation, both operands are converted to float and the result is float. For the other operations, if the operands are the same type, then the result is that type. If the operands are different numeric types and one operand is float, then the other is converted to float before the operation takes place, and the result is float. If one operand is integer type and the other is decimal type, then the integer is converted to decimal before the operation takes place, and the resulting value is decimal numeric type. In the following, **L** is the number of digits to the left of the decimal point in the result, **R** is the number of digits to the right of the decimal point, and **L1**, **R1**, **L2** and **R2** are the corresponding number of digits from the first and second operands.

For addition and subtraction:

L = greater of (**L1**, **L2**) + 2
R = greater of (**R1**, **R2**)

For multiplication and power:

L = **L1** + **L2**
R = **R1** + **R2**

For division:

L = **R1** + **R2**
R = greater of (**R1**, **R2**)

For modulus:

L = greater of (**L1**, **L2**)
R = 0

If the result is decimal or float numeric typeand **R** is zero, there is no decimal point in the result. For division, any extra digits on the right side of the result are truncated and no rounding occurs.

+ is the concatenation operator if both operands are character variables or literals. The result is a string that is the concatenation of the operands. The logical string of each operand are the strings that are concatenated.

=, **<>**, **<**, **>**, **<=** , and **>=** are the equal, not equal, less than, greater than, less than or equal, and greater than or equal operators, respectively. Note that **!=** may be used as an alternate form of the "not equal" operator. The operands must be either both numeric or both character. If the operands are character, their logical strings are compared. Different length strings are unequal. If the comparison is true, then the result is one. If the comparison is false, then the result is zero. The result is integer numeric type and the size is one.

and, **&**, **or**, and **|** are the logical and and or operators. **and** is equivalent to **&**. **or** is equivalent to **|**. Their operands are numeric. If one or both operands are exactly zero, then the result of the and operation is zero; otherwise the result is one. If both operands are exactly zero, then the result of the or operation is zero; otherwise the result is one. The result is integer numeric type and the size is one.

! is the reset size operator. The syntax of this operation is:

varexp **!** *numvarexp*

where *varexp* is a variable or expression and *numvarexp* is a numeric variable or numeric expression. The value of *numvarexp* defines the new size of the result of the reset size operation. If the left operand is numeric, then the result is decimal numeric type and the right operand defines the format. If the value of the right operand is fractional, the fractional amount is the number of digits to the right of the decimal point in the result. If the value of the right operand is non-fractional, then there is no decimal point in the result. The size is the left side of the result is the integer value of the right side. If the left operand is character, then the result is a character string. The logical string of the left side is either truncated on the right, or blanks are appended to create a string that is the length specified by the value of the right operand. Any fractional value is ignored.

Operator Precedence

When the precedence of operators is the same, all operations (except power) are performed from left to right. The power operation is performed from right to left. A listing of operator precedence is provided in the following table, grouped in order from highest precedence to lowest precedence.

Operator Precedence
( )
unary **–**, unary **+**, **sin**, **cos**, **tan**, **arcsin**, **arccos**, **arctan**, **exp**, **log**, **log10**, **sqrt**, **abs**, **formptr**, **lengthptr**, **size**, **length**, **int**, **form**, **float**, **char**, **fchar**, **lchar**, **squeeze**, **chop**, **trim**, **isnull**
!
**
* / %
+ –
< > <= >= = != <>
not
and &
or |

Rounding Numbers

Rounding may occur in certain statements such as the move statement. Rounding is the process of adjusting the numeric result so that the result fits the destination. Rounding is performed when the result contains more digits to the right of the decimal point than will fit into the destination.

If the number being rounded falls exactly between two possible results (there is no “nearest” number), then the number is rounded to the number with the greater absolute value.

Data Manipulation Statements

The data manipulation statements alter the content of numeric, character, and address variables and may also alter the equal, less, over, and eos flags.

The arithmetic statements (such as add, multiply, mod, and compare) typically set equal if the result of the operation is zero. If the result of the operation is non-zero, equal is cleared.

The arithmetic statements typically set less if the result of an operation is negative. If the result of an operation is zero or positive, less is cleared.

The arithmetic statements create an intermediate result and then move the intermediate result to the destination variable. The over flag is set if the most significant digit or sign of the intermediate result will not fit in the destination. When over is set, the values of the other flags are undefined. If the result of an arithmetic statement is not truncated, then over is cleared.

The operands of the arithmetic statements may be array variables or simple (non-array) variables. If the operands are simple variables, one arithmetic operation takes place.

If the first source operand is an array variable and the destination is a simple variable, then the arithmetic operation occurs between all elements of the source array. This is usually meaningful only for the add operation.

If the source and destination operands are arrays, then the arrays must be exactly the same size (both in number of dimensions and in number of elements in each dimension). One operation takes place for each element. equal is set only if the result of all operations is zero. less is set if any result is less than zero. over is set if any result is truncated.

If the operands for the add, subtract, compare, multiply, divide, or mod operations are different types of numeric variables, then they are converted as follows:

> If one operand is float, then the other operand is converted to float before the operation takes place.
>
> If one operand is integer and the other is number (form), then the integer is converted to number before the operation takes place.

Operations on character variables frequently use the logical string of the variable. If the form pointer is non-zero, the logical string includes all characters between the character pointed to by the form pointer and the character pointed to by the length pointer. The logical string is considered null if the form pointer is zero.

The logical string of a numeric variable includes all characters in the variable and leading blanks.

The logical length of a variable is the length of the logical string.

Program Flow Control Statements

When a DB/C program is compiled, a **.dbc** file is created. Each **.dbc** file is called a module. A module is considered to be the primary module if it is the first program executed (the default is the **answer.dbc** module) or if it is a program that has been the object of a chain statement. A module is considered to be a secondary module if it is loaded by the loadmod statement, the ploadmod statement, or the runtime preload mechanism.

Each program execution label is local to the module into which it is compiled unless the label has been specifically defined as an external (global) label. Local labels are unknown to other modules. External labels are recognized by other modules. A label definition is made external by the routine statement. A label reference is made external by the external statement.

Program flow begins at the first statement of the primary module and can continue into secondary modules by means of goto, branch, call, perform, or user-defined verbs that refer to external labels in other modules. Execution of the chain statement by either the primary module or a secondary module unloads all modules (except pre-loaded modules) and begins execution of a new primary module.

Each module contains data and program information. There is one data area for the primary module and for preloaded secondary modules. Other secondary modules may be associated with multiple copies (instances) of their data area. Only one of these data areas may be accessed at a time. The currently accessed data area is called the current instance of the secondary module. A program creates, switches between, and destroys instances of a secondary module with the loadmod and unload statements.

Access to variables is local to the current instance of a module with two exceptions. The first exception is access to common variables in secondary modules. Access to these variables permits access to the data area of the primary module. The second exception is address variables. Address variables may refer to data in any instance of any module. This access across modules is initiated by the call statement with parameters, by a user-defined verb with parameters, or by the getparm and loadparm statements.

Keyboard and Display Manipulation Statements

In a character mode environment, interactive input and output are accomplished with the keyin, display, and beep statements. The keyin and display statements contain lists of operands that define the actions that will take place with the key board and the terminal screen.

All printable characters are accepted from the keyboard except that some international characters are optionally supported. In addition, the Enter key, the Interrupt key, the Cancel key, function keys F1 through F20, and the extended function keys (Up, Down, Left, Right, Insert, Delete, Home, End, Page Up, Page Down, Tab, Back Tab, Backspace and Esc) are supported. All other keys pressed on the keyboard are ignored. The implementation of some or all of the keys is terminal dependent.

Pressing the interrupt key causes the equivalent of a stop operation to occur immediately.

The default window size is 25 lines of 80 columns. The window size may be reset to any size smaller or larger (up to the maximum size of the display screen).

The setendkey, clearendkey, and getendkey statements use numeric values to refer to specific keys. Integer values 1 through 255 represent the ASCII characters of the character set in use. The numeric values of the other ending keys are:

256 - 260	Enter, Esc, Backspace, Tab, Back Tab
261 - 270	Up, Down, Left, Right, Insert, Delete, Home, End, Page Up, Page Down
271 - 280	Shift+Up, Shift+Down, Shift+Left, Shift+Right, Shift+Insert, Shift+Delete, Shift+Home, Shift+End, Shift+Page Up, Shift+Page Down
281 - 290	Ctrl+Up, Ctrl+Down, Ctrl+Left, Ctrl+Right, Ctrl+Insert, Ctrl+Delete, Ctrl+Home, Ctrl+End, Ctrl+Page Up, Ctrl+Page Down
291 - 300	Alt+Up, Alt+Down, Alt+Left, Alt+Right, Alt+Insert, Alt+Delete, Alt+Home, Alt+End, Alt+Page Up, Alt+Page Down
301 - 320	F1 through F20
321 - 341	Shift+F1 through Shift+F20
341 - 360	Ctrl+F1 through Ctrl+F20
361 - 380	Alt+F1 through Alt+F20
381 - 406	Alt+A through Alt+Z

In a graphical user environment, interactive input and output can be accomplished with keyin, display, and beep in the same manner as in a character mode environment.

Printer Output Statements

Printer output operations direct data to a printer or a spool file. The maximum size of the print line is 400 characters for both printing and spooling to a print file.

Disk Input and Output Statements

Disk input and output consist of statements that cause data to be read and written from disk files. All data is read and written to the file with respect to logical records. The maximum record size is 65500 characters.

Five types of data files are supported by DB/C DX.

The first type of data file is the standard DB/C format file. Records consist of printable ASCII characters plus space and digit compression characters. Records are terminated by the end-of-record mark which is one character with the value hex FA. The end-of-file mark is one character with the value hex FB. Deleted records are replaced with the delete character which is hex FF.

The second type of data file is the portable text file. Use of this type of file is specified by adding the **data** operand to the file, ifile or afile variable definitions. Records are ASCII characters terminated with the LF (linefeed) character. There is no end-of-file character. Deleted records are replaced with the ASCII DEL character.

The third type of data file is the runtime operating system text file. Use of this type of file is specified by adding the **text** operand to the file, ifile or afile variable definitions. Records consist of characters that are valid text records for the runtime operating system. Depending on the operating system, different methods are used to specify end-of record, end-of-file, and deleted records.

The fourth type of data file is the user-defined file type. Use of this type of file is specified by adding the native operand to the file or ifile variable definitions.

The fifth type of data file is the binary file type. Use of this type of file is specified by adding the **binary** operand to the file, ifile or afile variable definitions. Records consist of binary data chunks. The size of these binary data chunks is determined by the **variable=** or **fixed=** operand. Typically the **fixed=** operand is used with the binary file type, but if the file size is not an even multiple of the record length, the **variable=** operand should be used. There is no end-of-record, end-of-file, or deleted record support.

The four access methods supported are: random, sequential, indexed sequential, and associative index. The file is considered indexed if the ifile declaration statement is used. The file is considered aimdexed if the afile declaration statement is used. The file is considered sequential if the **variable=** operand is specified on the file declaration statement. The file can be accessed by either sequential or random access when the **fixed=** operand is specified on the file declaration statement. In addition, an ifile and an afile can be accessed in a random or sequential mode.

The concept of file position is applicable to all access methods. The file position is a number that corresponds to a record within the file. In all cases it is the physical character position in the text file.

The position within the logical record for reads and writes is sequential left to right unless modified by the tab control codes. When a tab control code is encountered in the list, the position within the logical record is modified to the new tab position.

SQL Input and Output Statements

Access to Structured Query Language (SQL) databases is available with DB/C DX.

These three statements perform SQL operations: sqlexec, sqlcode, and sqlmsg.

Queue, Device, and Resource Variables

GUI programs use the queue, device and resource variables and operations extensively. In addition, custom soft ware can be written to interface with the queue, device, and resource operations.

Operating System Command Execution

Operating system commands may be executed directly as DB/C statements. Refer to Miscellaneous File Manipulation Statements for more information.

Communications

Communications operations allow tasks in the same system or in different systems to send and receive messages. Custom-written software may be installed to allow communications.

Syntax Abbreviations

Throughout the manual, these abbreviations are used:

adrvar	address form of a variable
afile	afile variable
array	character or numeric array variable
cadrvar	address form of a character variable
cfile	comfile variable
charexp	character variable, character literal, or character expression
charlit	character literal string
charvar	character variable
chrarray	character array variable
class	class definition label
dcon	equate label or decimal constant
device	device variable
equate	equate label
exp	character variable, numeric variable, character literal, numeric constant, character expression, or numeric expression
expression	character or numeric expression
file	file variable
hexcon	equate label or decimal, hexadecimal, or octal constant
ifile	ifile variable
image	image variable
label	label
lblvar	label variable
lstvar	list variable
method	label of a class routine
numarray	numeric array variable
numexp	numeric variable, numeric literal, or numeric expression
numlit	numeric literal string
numvar	numeric variable
objvar	object variable
pfile	pfile variable
prep	preposition or comma
queue	queue variable
resource	resource variable
var	character or numeric variable
variable	any variable except a label variable
varvar	typeless (var) address variable

Compiler Directives

Compiler directives are a class of instructions which control the compilation process.

The compiler directives do not affect and are not affected by the flags.

Conditional Compiler Directives

label	**%if**	*hexcon1 operator hexcon2*
label	**%ifz**	*list1*
label	**%ifnz**	*list1*
label	**%ifdef**	*equate*
label	**%ifdef**	*var*
label	**%ifdef**	*list2*
label	**%ifndef**	*equate*
label	**%ifndef**	*var*
label	**%ifndef**	*list2*
label	**%iflabel**	*list3*
label	**%ifnlabel**	*list3*
label	**ifeq**	*hexcon1, hexcon2*
label	**ifne**	*hexcon1, hexcon2*
label	**ifgt**	*hexcon1, hexcon2*
label	**ifng**	*hexcon1, hexcon2*
label	**iflt**	*hexcon1, hexcon2*
label	**ifnl**	*hexcon1, hexcon2*
label	**ifge**	*hexcon1, hexcon2*
label	**ifle**	*hexcon1, hexcon2*
label	**%else**	
label	**%elseif**	*hexcon1 operator hexcon2*
label	**%endif**	
label	**xif**	

All conditional compiler directives that have % as the first character may also be used with **#** as the first character (in place of the %)
ifs and **ifnz** works like **%ifnz**, except they should only be used with **xif**.
ifc and **ifz** works like **%ifz**, except they should only be used with **xif**.
label is optional
hexcon1 is the first operand
hexcon2 is the second operand
operator is one of these: **< > <= >= = <>**
list1 is a list of *hexcon* operands
equate is an equate label
var is a data variable label
list2 is a list of equates and data variable labels
list3 is a list of program label names

The conditional compiler directives determine if the program lines within the scope of the construct will be compiled.

The result of the evaluation of the operands of the **%if**.., **if**.. and **%elseif** statements determines the conditional compilation. If the condition associated with one of these statements is satisfied, the program lines within the scope of the directive are compiled. If the condition is not satisfied, the program lines within the scope of the directive are not compiled.

If the preceding **%if**.. or **if**.. directive statement condition was not satisfied, the **%elseif** statement causes the following program lines to be compiled if the condition of the **%elseif** statement is satisfied.

The **%else** statement reverses the current conditional compilation status. If an **%else** directive is encountered and no previous condition has been satisfied, then the lines of code within the scope of the

%else directive are compiled. If an **%else** directive is encountered and a previous condition has been satisfied, then the lines of code within the scope of the **%else** directive are not compiled.

The **%endif** statement terminates the most recent conditional compilation directive action.

The **xif** statement ends the scope of all previous if conditional compilation statements that do not begin with the % character. That is, xif does not apply to or affect the scope of **%if**, **%ifdef**, etc. It only applies to the ifeq, ifne, etc. statements. This action is modified by the **-8** and **-9** compiler command line parameters.

The condition is satisfied as follows:

Directive	Satisfied Condition
`%if`	if the comparison operation (specified by the operator) on the first and second operands is true
`%ifz`	if any hexcon in the list is zero
`%ifnz`	if any hexcon in the list is non-zero
`%ifdef`	if any equate or variable in the list is previously defined in the program
`%iflabel`	if any equate or variable in the list is not previously defined in the program
`%ifnlabel`	if any program label in the list is not previously defined in the program
`%elseif`	if the comparison operation (specified by the operator) on the first and second operands is true
`ifeq`	if the first operand is equal to the second operand
`ifne`	if the first operand is not equal to the second operand
`ifgt`	if the first operand is greater than the second operand
`ifng`	if the first operand is not greater than the second operand
`iflt`	if the first operand is less than the second operand
`ifnl`	if the first operand is not less than the second operand
`ifge`	if the first operand is greater than or equal to the second operand
`ifle`	if the first operand is less than or equal to the second operand

define

label **`define`** *any-string*
label **`define`** `"`*any-string*`"`

label is required
any-string is a series of characters

The define statement assigns a string of characters to a label. Whenever the label is encountered in the source file in an operand position, it is replaced by the string of characters any-string. any-string may include previously defined labels. Recursive expansion will result in an error.

Quotation marks (`"`) must enclose any-string if any-string includes one or more blanks. To include a quotation mark character in the string, use the pound sign (**#**) as a forcing character. Similarly, to include a pound sign character in the string, use the pound sign forcing character.

equate

label **`equate`** *charlit*
label **`equate`** *hexcon*

`equ` may be used in place of **`equate`**

label is required
charlit is a one-character literal string
hexcon is a constant or equate label that represents a number from 0 to 65500

The equate statement assigns an integer value to a label. Whenever the label is encountered in the program, it is replaced by the integer value defined in the corresponding equate statement. In this way, an equate label can be used in place of a decimal, octal, or hexadecimal constant.

If the operand field is a single-character literal, the decimal value of the character is assigned to the equate label.

include

label **`include`** *filename*

`inc` may be used in place of **`include`**

label is optional
filename is the name of a source file

The include statement inserts the contents of another file into the program. When the compiler encounters an include statement, it replaces the statement with the contents of the specified source file. Those contents are then compiled as if they actually existed in the source program.

If no extension is specified for filename, **.txt** is assumed.

Includes may be nested up to sixteen levels deep.

Certain compiler command line parameters may modify the include file name.

liston, listoff

```
liston
listoff
```

The liston and listoff directives control whether the listing of the source program is displayed during the compilation process.

The source program is only listed on screen when the **-d** option is specified on the compiler command line and liston is in effect. liston is automatically in effect at the beginning of compilation. listoff prevents the display of subsequent lines of code, while liston permits the display of subsequent lines of code.

The liston and listoff directives have no effect on the compilation process; they simply turn the listing mechanism on and off.

Definition Statements

There are several types of definition statements: variable definitions, class definitions, method definitions, routine definitions and user-defined verb definitions.

Variables are defined by variable definition statements. Variable definition statements may be located at any place in the program. The only restriction is that the variable definition statement for a variable must precede any executable statement which references the variable. There is one exception to this general rule. A parameter used in a routine or lroutine statement may be declared immediately after the routine or lroutine statement.

Certain types of variables allow arrays of those variables to be defined. Arrays are defined with one, two, or three dimensions. Each dimension may contain up to 32767 elements.

Character Variables

label **`char`** *dcon*
label **`char`** *dcon* `,` **`initial`** *value*
label **`char`** *dcon* `[`*array-spec*`]`
label **`char`** *dcon* `[`*array-spec*`]` `,` **`initial`** *values*
label **`init`** *list*

`dim` and **`character`** may be used in place of **`char`**
`(`*array-spec*`)` may be used in place of `[`*array-spec*`]`

label is the name of the variable
dcon is the maximum length of the variable
`[`*array-spec*`]` is one of `[`*dcon*`]`, `[`*dcon* `,` *dcon*`]` or `[`*dcon* `,` *dcon* `,` *dcon*`]`
value is a literal value enclosed in double quotation marks
values is a comma delimited list of literal values enclosed in double quotation marks
list is a list of character literals and decimal, octal, and hexadecimal constants

A character variable definition statement defines a character variable.

A character variable contains a character string. A form pointer, a length pointer, and a maximum length are associated with each character variable. The valid values of the form pointer, the length pointer, and the maximum length are 0 through 65500. The value of the form pointer is always less than or equal to the value of the length pointer. The value of the length pointer is always less than or equal to the maximum length.

For the char statement, the *dcon* operand specifies the maximum length of the variable. If the **initial** *value* syntax is not specified, the initial value of a char variable is blanks with the form pointer and length pointer set to zero. If **initial** *value* is specified, the form pointer is set to one, the length pointer is set to the number of characters in the *value* literal, and the *value* literal is placed in the variable followed by blanks.

The `[`*array-spec*`]` forms of the char statement define an array of character variables. The **initial** *values* syntax form allows arrays to be initialized in the declaration statement. *values* is a comma delimited list of literal values, each enclosed in quotation marks. Arrays are initialized in row-major order.

An init statement defines a character variable with an initial value made by appending the literals and constants from the list. The total length (in characters) of the literals and constants in the list is the maximum length of the variable. The form pointer is set to one. The length pointer is set to the maximum length.

The list in an init statement may be continued to the next program line by means of a colon. A character variable defined by an init statement may contain up to 8192 characters.

Numeric Variables

label	**`num`**	*dcon1*
label	**`num`**	*dcon1*`.`*dcon2*
label	**`num`**	*numlit*
label	**`num`**	*dcon1*`[`*array-spec*`]`, *initial values*
label	**`num`**	*dcon1*`.`*dcon2*`[`*array-spec*`]`, *initial values*
label	**`integer`**	*dcon1*
label	**`integer`**	*numlit*
label	**`integer`**	*dcon1*`[`*array-spec*`]`, *initial values*
label	**`float`**	*dcon1*
label	**`float`**	*dcon1*`.`*dcon2*
label	**`float`**	*numlit*
label	**`float`**	*dcon1*`[`*array-spec*`]`, *initial values*
label	**`float`**	*dcon1*`.`*dcon2*`[`*array-spec*`]`, *initial values*

`number` and **`form`** may be used in place of **`num`**
`int` may be used in place of **`integer`**
`(`*array-spec*`)` may be used in place of `[`*array-spec*`]`

label is the name of the variable
dcon1 is the number of digits left of the decimal point
dcon2 is the number of digits right of the decimal point
numlit is the initial value
`[`*array-spec*`]` is one of `[`*dcon*`]`, `[`*dcon*`,` *dcon*`]` or `[`*dcon*`,` *dcon*`,` *dcon*`]`
values is a comma delimited list of numeric values

The numeric variable definition statements define numeric variables. *dcon1* is optional in the *dcon1*`.`*dcon2* forms of these statements. When *dcon1* is not specified, it defaults to zero. When *dcon2* is not specified, it defaults to zero. When *dcon2* is zero, there is no decimal point.

A numeric variable always contains a valid DB/C number string. The number is right justified and leading zeros are suppressed.

The number statement defines a decimal numeric variable that contains an exact representation of a number with the number of digits specified by *dcon1* and *dcon2*.

The integer statement defines an integer numeric variable. The initial value of an integer variable is zero. Whenever an integer variable is displayed, printed, or written to a file, it is first converted to a string of digits with width specified by the *dcon1* operand. If the width is too small to represent the number, the result is undefined.

The float statement defines a float numeric variable. The initial value of a float variable is zero. Whenever a float variable is displayed, printed, or written to a file, it is first converted to a string of digits (and a decimal point if *dcon2* is non-zero) with *dcon1* digits left of the decimal point and *dcon2* digits right of the decimal point. If *dcon1* is too small to contain the value, the result is undefined. If *dcon2* is too small, the number is truncated to fit.

The forms of the num, integer, and float statements that have a *numlit* operand define the size and initial value of the numeric variable. The non-*numlit* forms all set the initial value of the variable to zero.

The `[`*array-spec*`]` forms of the numeric variable definition statements define arrays of numeric variables. The **initial** *values* syntax is optional and allows arrays to be initialized in the declaration statement. *values* is a comma delimited list of decimal constant values. Arrays are initialized in row-major order.

List Variables

label **`list`**
label **`list with names`**
`listend`
label **`varlist`** *list*

label is the list variable name
list is a list of character variables, numeric variables, literals, arrays, array elements, list variables, file variables, and special variables

The list statement associates a group of data variables with a single label by defining a list variable. The variables specified immediately after the list statement are included in the list variable. The listend statement indicates the end of the list of variables. The list followed by listend statement pairs may be nested.

The **with names** form of the list statement allows the getname statement to provide the names of the list variable and of all the variables contained in the list. See the getname statement.

The varlist statement defines a list variable. The list of variables associated with varlist is a list of previously defined variables that do not have to be contiguous.

list can contain variables of any type including file, ifile, afile, queue, record, etc. However, an attempt to use a list as an operand in a statement that does not allow variables of the types included in the list will result in an E 557 error. For example, the clear statement does not allow file type variables as operands. Therefore, a list that contains a file cannot be cleared.

Record Variables

label1 **`record`**
label1 **`record with names`**
label2 **`record definition`**
label1 **`record like`** *label2*
label1 **`record like`** *label2* **`with names`**
`recordend`

label1 is the name of the record being declared
label2 is the name of a record prototype definition

See also: List Variables, getname

The record statement defines a list variable. The **record like** forms of the record statement are not used in conjunction with the recordend statement. The other record statements are followed by zero or more variable definition statements which are then immediately followed by a recordend statement.

The forms of the record statement without the **definition** or **like** keywords create a list of variables as an actual data item. The record and recordend pair of statements works exactly the same as a list and listend statement pair, except the names of all variables contained within the record and recordend pair are prefaced with *label1* followed by a period. These forms of the record and recordend statement pair may be contained within a list and listend statement pair and may contain a list and listend statement pair. Partial overlap with a list and listend pair is not allowed.

The form of the record statement with the **definition** keyword is used to create a record prototype. The variables specified immediately after the record definition statement are included in the record prototype definition. The recordend statement indicates the end of the prototype definition. All variables contained within the record definition followed by recordend scope are not actually created, but are used as models. No actual data item of the name *label2* exists. This form of the record and recordend statement pair may not be contained within a list and listend statement pair, but may contain a list and listend statement pair. Partial overlap with a list and listend statement pair is not allowed.

In all cases, record definition and recordend statement pair may not be nested.

The forms of the record statement with the **like** keyword are used for data definition by prototype. The record like statement creates a record using the prototype (defined by a record definition statement) whose label matches the label following the like operand. The record definition statement must precede the record like statement in the source program. A record that is declared by the record like form is an actual data item. Each variable in the record will have *label1* followed by a period attached to the beginning of its name. A recordend statement is not used in conjunction with a record like statement. A record like statement may be located anywhere that a normal data definition statement may be located, except not within another record and recordend statement pair.

The **with names** forms of the record statement allow the getname statement to provide the list variable name (*label1*) and the name of all variables contained in the list. See the getname statement.

File Variables

label	**file**	*operand-list*
label	**ifile**	*operand-list*
label	**afile**	*operand-list*
label	**pfile**	
label	**comfile**	

label is the file variable name
operand-list is a comma delimited list of operands

The file statement defines a random and sequential file variable. The file statement operand list may be empty or may consist of one or more of the following operands:

fixed=*dcon* **fixed=***numvar*	defines the file to consist of fixed length records. If *dcon* is specified, it is the record length. If *numvar* is specified, its value is the record length at the time the file is opened. The value of *numvar* is truncated if it is fractional. **fixed** may be abbreviated to **fix**. The largest value allowed for *dcon* is 65500.
variable=*dcon* **variable=***numvar*	defines the file to consist of variable length records. If *dcon* is specified, it is the maximum record length. If *numvar* is specified, its value is the maximum record length at the time the file is opened. The value of *numvar* is truncated if it is fractional. **variable** may be abbreviated to **var**. If neither **fixed** nor **variable** is specified, **variable=256** is the default. The largest value allowed for *dcon* is 65500.
compressed	defines the file to consist of space and number compressed records. **compressed** is mutually exclusive with **fixed**, **data**, **native**, and **binary**. **compressed** may be abbreviated to **comp**. If variable is specified or assumed, and **compressed** and **uncompressed** are not specified, **compressed** is the default.
uncompressed	defines the file to consist of records which are not compressed. **uncompressed** may be abbreviated to **uncomp**. If **fixed** is specified, **uncompressed** is assumed.
standard	defines the file as a standard DB/C-type file. If **standard**, **text**, **native**, and **binary** are not specified, **standard** is the default. **standard** is mutually exclusive with **text**, **data**, **crlf**, **native**, and **binary**.
text	defines the file to be compatible with text files of the runtime operating system. **text** is mutually exclusive with **standard**, **data**, **crlf**, **native**, and **binary**.
data	defines the file to be an ASCII data file with LF (linefeed) as the end-of-record character, and DEL as the deleted space character. There is no end-of-file character. **data** is mutually exclusive with **standard**, **text**, **crlf**, **native**, and **binary**.
crlf	defines the file to be an ASCII data file with CR (carriage return), LF (linefeed) as the end-of-record marker. **crlf** is mutually exclusive with **standard**, **text**, **data**, **native**, and **binary**.
native	defines the file to be used with a certain file type unique to an operating system and runtime. This option is only valid when used with a specially configured runtime. **native** is mutually exclusive with **standard**, **text**, **data**, **crlf**, and **binary**.
binary	allows access to binary data as fixed records. **binary** is mutually exclusive with **standard**, **text**, **data**, **crlf**, and **native**.
static=*dcon*	defines the size of the buffer used for sequential reads and writes from the file. *dcon* is a number from 1 to 8. 1 denotes the smallest buffer and 8 denotes the largest buffer. Buffering is only done for files opened in exclusive mode.
cobol	enables a program to read and write files that are compatible with the COBOL programming language. This affects the action of read and write on numeric variables.

When writing a numeric variable to a COBOL-type file, any space or minus sign is written as the character 0 and the decimal point is not written. If the value of the field is negative, the least significant digit of the variable is altered before it is written to disk. The last character is altered to `}` if the far right digit was **0**. The last character is altered to a letter between **J** and **R** if the far right digit was **1** through **9**, respectively.

When reading a numeric variable from a COBOL-type file, the character **0** is converted to a blank character (or a minus sign if the **0** is in the most significant digit position). The decimal point is implied by the receiving field. If the far right character is `}`, it is changed to **0** and the value of the field is negative. If the far right character is letter **J** through **R**, the value of the field is negative, and the last character is altered to a digit between **1** and **9**, respectively. If the far right character is `{`, it is changed to **0**. If the far right character is letter **A** through **I**, it is changed to a digit between **1** and **9**, respectively. If a number cannot fit into the variable, a format error occurs. This error occurs when the number is negative and a character other than **0** is contained in the most significant digit position.

Note that the records read and written with the **cobol** operand in effect are shorter than the records read and written with the same variable list without the **cobol** operand in effect. The number of bytes shorter is equal to the number of numeric fields in the list that contain a decimal point. This shorter length must be reflected in the **fixed** and **variable** operands.

`dynamic` is ignored.

`overlap` is ignored.

`increment` is ignored.

The ifile statement defines an indexed file variable. The operand list may include the same operands used with the file statement. Additionally, the ifile operand list may contain one or both of these operands:

`keylength=`*dcon*
`keylength=`*numvar* defines the key length of the key associated with this file. The maximum key length is 255. If the second form is used, *numvar* is a previously defined numeric variable. Upon opening the file, the value in numvar defines the key length of the ifile. If **keylength** is not specified, then the key length used is that specified by the index utility when the index was created. **keylength** may be abbreviated to **keylen**.

`duplicates` indicates that duplicate keys will be allowed in the file. If neither **duplicates** nor **noduplicates** is specified, then the original duplicate specification in the index file controls whether or not duplicates are allowed. **duplicates** may be abbreviated to **dup**. **duplicates** is mutually exclusive with **noduplicates**.

`noduplicates` indicates that duplicate keys will not be allowed in the file. **noduplicates** is mutually exclusive with **duplicates**. **noduplicates** may be abbreviated to **nodup**.

The afile statement defines an associative index file variable. The operand list may include the same operands used in the file statement.

The pfile statement defines a print file variable. A print file variable defines the destination of the print image created by the print, splopen, and splclose statements.

The comfile statement defines a communications file variable.

Classes, Object and Inherited Variables

label `class` `definition,` *operand-list*
label `endclass`
label `class`
label `class` `module=`*charlit*
label `method`
label `object`

The class statement with definition defines a class and is the first statement of the group of statements that is the body of a class. The body of a class ends with the endclass statement. The class definition and endclass statement pair can not be nested. There may be multiple class and endclass statement pair class definition groups in a compile unit.

The operands of the class definition statement are optional. Each may be specified once. The operands are:

`make=`routine-label
`destroy=`routine-label
`parent=`class-name

The operands are separated by commas. If no operands are specified, the preceding comma must be omitted.

make and **destroy** identify the routines that are automatically called when an object is created or destroyed. These routines must be defined by a routine statement inside of the class definition.

parent identifies the class that is the ancestor of this class. Methods and variables are inherited from the parent and from all ancestors of the parent.

The class statements without **definition** define a class whose body is defined elsewhere. A class statement without **module** defines a class whose definition and body are contained later in the same compile unit. When **module** is present, the class definition and body are in another compile unit. The *charlit* value of the **module** operand is the name of the compiled object file that contains the class definition of this class. If a file extension is not specified in *charlit*, then **.dbc** is assumed. The *charlit* value may be up to 31 characters in length. Class statements without the **definition** keyword are not used in conjunction with an endclass statement.

The method statement declares a method whose body is defined elsewhere. A method declared by a method statement may be called by subsequent call statements. The body of the method (the routine or lroutine statement that defines it) may be defined in this compile unit or in another compile unit. The method statement must be used even if the method is defined in the same compile unit.

The object statement defines an object variable. An object variable is instantiated explicitly by the make statement or implicitly by execution of a user-defined verb that has the `!`**make** or `!`**transient** operands in its definition.

Inheritable variables are defined by preceding their operands with `&`. All types of variables except global, common, label, record, list, and varlist type variables may be defined as inheritable variables. Inheritable variables must be defined inside a class definition, but outside of any routines in that class.

Inherited variables are defined by preceding their operands with `&&`. Inherited variables need not contain variable length, format, array size, and most other information. They only need to contain variable type, array dimension information, and address variable designation information. Inherited variables must be defined inside a class definition, but outside of any routines in that class. Inherited variables can only be specified in classes that have a parent that contains inheritable variables with the same name and type. Inherited variables are also inheritable.

Special Variables

label	**`device`**	
label	**`resource`**	
label	**`image`**	*operand-list*
label	**`queue`**	*operand-list*

label is the variable name
operand-list is a list of operands

The device statement defines a device variable.

The resource statement defines a resource variable.

The image statement defines an image variable. The operand list contains the following operands:

`h=`*dcon*	defines the horizontal size (in pixels) of the image stored in the image variable. This operand is required.
`v=`*dcon*	defines the vertical size (in pixels) of the image stored in the image variable. This operand is required.
`colorbits=`*dcon*	defines the number of bits contained in each pixel. This operand is optional. Valid values are 1, 4, 8, 16, and 24. The default is runtime dependent.

The queue statement defines a queue variable. The operand list may be empty or may contain one or both of the following operands:

`entries=`*dcon*	defines the maximum number of entries contained in the queue variable. The default is 32.
`size=`*dcon*	defines the size (in characters) of each queue entry. The default is 32.

Address Variables

label	**char**	@, **initial** *value*
label	**char**	[*array-spec*]@, **initial** *values*
label	**char**	@[ ], *initial value*
label	**char**	@[,], *initial value*
label	**char**	@[,,], *initial value*
label	**num**	@, *initial value*
label	**num**	[*array-spec*]@, *initial values*
label	**num**	@[ ], *initial value*
label	**num**	@[,], *initial value*
label	**num**	@[,,], *initial value*
label	**list**	@, *initial value*
label	**file**	@, *initial value*
label	**ifile**	@, *initial value*
label	**afile**	@, *initial value*
label	**comfile**	@, *initial value*
label	**pfile**	@, *initial value*
label	**object**	@, *initial value*
label	**device**	@, *initial value*
label	**resource**	@, *initial value*
label	**image**	@, *initial value*
label	**queue**	@, *initial value*
label	**var**	@, *initial value*
label	**var**	[*array-spec*]@

The **initial** *value* and **initial** *values* operands are optional. If not specified, the preceding comma must be omitted.

character and **dim** may be used in place of **char**
number, **form**, **integer**, **int**, and **float** may be used in place of **num**

label is the variable name

The @ form of an address variable definition statement defines an address variable of the type specified by the operation. With regard to address variables, all numeric types are considered to be the same.

The [*array-spec*]@ form defines an array of address variables. The @[ ], @[,], and @[,,] forms define an address variable that points to a one, two, or three-dimensional array of the type specified by the operation.

The var statement defines a special type of address variable called a typeless address variable. Typeless address variables can only be used in moveadr, loadadr, storeadr, type, clearadr, movelv, movevl, and testadr statements.

Address variables are assigned values by statements like moveadr and by the call statement with parameters. When a statement makes reference to an address variable, the variable that is pointed to is the variable on which the statement actually operates. If a reference is made to an address variable that does not have a value assigned to it, a runtime error occurs.

Address variables are assigned initial values if **initial** is specified. The *value* operand is the name of a non-address variable of the same type that is defined before the statement with the initial operand. *values* is a comma delimited list of non-address variables of the same type. For typeless address variables, the label in the *value* operand does not need to match in type.

Global Variables

label **`gchar`** *char-var-spec*
label **`gnumber`** *number-var-spec*
label **`ginteger`** *integer-var-spec*
label **`gfloat`** *float-var-spec*
label **`gobject`**
label **`gdevice`**
label **`gresource`**
label **`gimage`** *image-var-spec*
label **`gqueue`** *queue-var-spec*
label **`gpfile`**

`gnum` and **`gform`** may be used in place of **`gnumber`**
`gcharacter` and **`gdim`** may be used in place of **`gchar`**

char-var-spec is char variable definition operands
number-var-spec is number variable definition operands
integer-var-spec is integer variable definition operands
float-var-spec is float variable definition operands
image-var-spec is image variable definition operands
queue-var-spec is queue variable definition operands

The global variable definition statements define global variables. A global variable persists across program chaining. In all other respects a global variable is identical to a non-global variable. The global variable type must match across program chaining, but the size is ignored if the global variable already exists. Global arrays are allowed. Any initial value or values are ignored if the global variable already exists.

Common Variables

label	**`char`**	`*`*operand*
label	**`init`**	`*`*operand*
label	**`num`**	`*`*operand*
label	**`integer`**	`*`*operand*
label	**`float`**	`*`*operand*
label	**`file`**	`*`*operand*
label	**`ifile`**	`*`*operand*
label	**`afile`**	`*`*operand*
label	**`comfile`**	`*`
label	**`pfile`**	`*`

`character` and **`dim`** may be used in place of **`char`**
`number` and **`form`** may be used in place of **`num`**
`int` may be used in place of **`integer`**

operand is the size and optional `[`*array-spec*`]` operand as defined previously for each data definition statement

When a program begins execution of another program, data items may be passed unmodified from the first program to the second. Such data items, passed from one program to its successor, are called common variables. A variable is defined as a common variable by a form of the data definition statement which contains an asterisk (`*`) immediately preceding the operand or operands.

For common data items to be successfully passed from one program to the next, certain rules must be observed. All common data must precede any non-common data. All common data items in the new program must match all data items (common or non-common) in the primary program, in the same order. To match, data items must be of the same type: character variables must be of the same declared length, numeric variables must be of the same class and must have been declared with the same number of integral and fractional digits, and arrays must have the same number of dimensions, be of the same size in each dimension, and be of the same elemental type of the same size.

If the primary module has fewer variables (common and non-common) than the successor program has common variables, the additional common data items are not treated as common.

If the common variable is encountered in the primary module when the program is being loaded (that is, chained to from another program), then the value of the variable remains unchanged. If the common variable is encountered in a secondary module (that is a module loaded by the loadmod statement), then the common variable refers to a variable in the primary module.

Common variables may be used like regular variables, except they may not be used in varlist variables in pre-loaded modules.

Label Variables

label **`label`** `@`

See also: movelabel, movelv, movevl, loadlabel, storelabel, routine, lroutine, getparm, loadparm

The label statement defines a label variable. A label variable defines a program execution label that may be changed at runtime. All program flow control statements that refer to an execution label may refer to a label variable.

verb, cverb

label **`verb`** *proto-lis*
label **`cverb`** *proto-list*

proto-list is a comma delimited list of operand prototypes

See also: call, ccall, class, destroy, getparm, loadparm, make, resetparm

The verb and cverb statements define user-defined verbs. A user-defined verb is a verb that may be used like any regular executable verb, except the user-defined verb is implemented by user-provided DB/C or C code.

When a user-defined verb is encountered in a source program, the compiler checks its syntax against the syntax defined by the *proto-list*. If the syntax is incorrect, a compiler error is generated. If the syntax is correct for a user defined verb defined by a non-object oriented verb statement, then the compiler generates a call statement (possibly with parameters). If the syntax is correct for a user-defined verb defined by a cverb statement, then the compiler generates a call to the cverb function.

When a user-defined verb is encountered in the source program with a **`:`***objvar* appended to the user-defined verb, then the user-defined verb is calling a method in the class from which the *objvar* was created. If there is no method of that name in the appropriate class (or inherited class), then an E 551 error occurs.

When a user-defined verb is executed that was defined with a class name (the second form of the verb statement) causes an implicit make statement to execute before the call and an implicit destroy statement to execute after the call statement. The implicit make, call and destroy all reference an unnamed, automatically created and destroyed object variable.

There are three operand prototypes that correspond with the three types of operands:

1. The prototype of a positional operand is:

   `#`*type*

2. The prototype of a non-positional operand is:

   `=`*type*

3. The prototype of a keyword operand is any of these:

   keyword
   *keyword***`=`***type*
   *keyword***`=`***type***`:`***type*
   *keyword***`=`***type***`:`***type***`:`***type*
   *keyword***`=`***type***`:`***type***`:`***type***`:`***type*
   *keyword***`=`***type***`:`***type***`:`***type***`:`***type***`:`***type*

The keyword syntax is the same as any other DB/C label. It must begin with a letter and it may contain letters, digits and certain other characters.

When a user-defined verb is encountered in a source program, the compiler checks its syntax against the syntax defined in *proto-list*. If the syntax is incorrect, a compiler error is generated.

The verb statement defines an object-oriented user-defined verb if the first operand of *proto-list* is one of these:

`!make`
`!make(`*class-name***`)`**
`!method`
`!destroy`
`!transient(`*class-name***`)`**

Execution of a user-defined verb defined by either of the two **`!make`** forms causes the compiler to generate a make operation, followed by a call to the routine specified by the name of the user verb.

If the class name is not specified (the first form of **!make**), then the user-defined verb must be used like this:

verb objvar `(`*class-name*`)` `,` *parameter-list*

If the class name is specified in the verb statement, then the user-defined verb must be specified like this:

verb objvar `,` *parameter-list*

In all cases, *parameter-list* is optional. If it is not specified, the comma preceding it is omitted.

The **!method** and **!destroy** forms require the user-defined verb to be specified like this:

verb objvar `,` *parameter-list*

Execution of a user-defined verb defined with **!method** causes the compiler to generate a call to the method whose name is the same as the verb. Execution of a user-defined verb defined with **!destroy** causes the compiler to generate a call to the method whose name is the same as the verb, followed by a destroy operation.

The **!transient** form requires that the user-defined verb be specified normally, with the optional *parameter-list*. It causes the compiler to generate a make operation, a call to the method and a destroy operation, all using a temporary unnamed object variable.

The values for type are specified in this table:

Value	Meaning
`array`	any array variable
`afile`	an afile variable
`any`	any type of variable
`carray`	a 1, 2 or 3 dimensional character variable array
`carray1`	a 1 dimensional character variable array
`carray2`	a 2 dimensional character variable array
`carray3`	a 3 dimensional character variable array
`cnvar`	a character or numeric variable
`cnvarlit`	a character or numeric variable or literal
`comfile`	a comfile variable
`cvar`	a character variable
`cvarlit`	a character variable or literal
`device`	a device variable
`file`	a file variable
`ifile`	an ifile variable
`image`	an image variable
`list`	a list or varlist variable
`label`	a program label
`nvar`	a numeric variable
`nvarlit`	a numeric variable, literal or decimal constant

continued on next page

Value	Meaning
`narray`	a 1, 2 or 3 dimensional numeric variable array
`nvarray1`	a 1 dimensional numeric array variable
`nvarray2`	a 2 dimensional numeric array variable
`nvarray3`	a 3 dimensional numeric array variable
`pfile`	a pfile variable
`object`	an object variable
`queue`	a queue variable
`resource`	a resource variable
`var`	any single valued variable
`varlit`	any single valued variable or literal

Positional operand prototypes must precede the other prototypes. Positional operand prototypes define required positional operands that must appear as operands of a user-defined verb. The order of the operands must match the order of the prototypes. Positional operands are compiled as parameters on the generated call or ccall statement.

There may only be one non-positional prototype in the list of prototypes. This prototype defines the type or types of variables that may be included in the list of user-defined verb operands. The non-positional operand may be before, after, or between key word operands in the operand list.

Any number of different keyword operand prototypes may be defined by a verb or cverb statement. Keyword operand prototypes without the equal sign define keywords that may be found in the operand list. Keyword operand prototypes with the equal sign define keywords that may be found in the operand list, but that must be followed by an equal sign and one or more colon delimited variables that match the variable types specified in the prototype.

Executable Statements

Executable statements are a class of instructions that cause actions to occur at runtime.

add

label	**`add`**	*numexp1 prep numvar3*
label	**`add`**	*numexp1 prep numexp2* **`giving`** *numvar3*
label	**`add`**	*numexp1 prep numarray3*
label	**`add`**	*numarray1 prep numvar3*
label	**`add`**	*numarray1 prep numarray3*
label	**`add`**	*numarray1 prep numarray2* **`giving`** *numarray3*

numexp1 is the first source operand
numexp2 is the second source operand
numvar3 is the destination operand
numarray1 is the first source operand
numarray2 is the second source operand
numarray3 is the destination operand

Flags affected: equal, less, over

If the first format is used, then the source operand is added to the destination operand and the result is placed in the destination operand.

If the second format is used, then the first source operand is added to the second source operand and the result is placed in the destination operand.

If the third format is used, the source operand is added to each element of the destination operand.

If the fourth format is used, all elements of the source array are added together and the result is placed in the destination operand.

If the fifth format is used, each element of the source array is added to the corresponding element in the destination array.

If the sixth format is used, each element of the first operand array is added to the corresponding element of the second source operand array, and the result is placed in the corresponding element of the destination array.

Rounding takes place when the intermediate result is moved to the destination.

and

label **`and`** *charvar1 prep charvar2*
label **`and`** *numlit prep charvar*
label **`and`** *numvar1 prep numvar2*

charvar1 is the source operand
numlit is the source operand
numvar1 is the source operand
charvar2 is the destination operand
charvar is the destination operand
numvar2 is the destination operand

Flags affected: equal, eos

See also: or, not, xor, rotate

For the first and second formats of the and statement, the bitwise and operation is performed on the form pointed characters from the source and destination operands. The result is stored in the form pointed position in the destination operand. If a decimal, hexadecimal, or octal constant is used as an operand, the character represented by that character code is used. If the result is binary zero, equal is set. If either string is null, eos is set and no changes are made.

The result of the and operation is determined by comparing the bits in each operand:

0 AND 0 evaluates to 0
0 AND 1 evaluates to 0
1 AND 0 evaluates to 0
1 AND 1 evaluates to 1

For the third format of the and statement, the source and destination operands are converted to 32 bit integers and the operation is performed. The result is moved to the destination operand. If the result is zero, equal is set. Otherwise, equal is cleared.

append

label **`append`** *exp prep charvar*

exp is the source operand
charvar is the destination operand

Flags affected: eos

The append statement appends a string to a character variable. The logical string of the source is moved to the destination beginning at the position immediately following the form pointed character of the destination variable.

The move continues until the maximum length of the destination variable is reached or until the end of the source logical string is encountered. The form pointer and logical length pointer of the destination are set to point to the last character moved.

If characters are lost to truncation because the source does not completely fit into the destination, eos is set.

If the source operand is null, no change is made to the destination.

beep

label **`beep`**

Flags affected: none

See also: sound

The beep statement causes a beep to sound.

branch

label **`branch`** *numexp prep list*

numexp is the index
list is a comma delimited list of program execution labels and program label variables

Flags affected: none

The branch statement causes program execution to continue with one of the entries in *prog-label-list* based on the value of the index. Program execution continues at the statement specified by the Nth label in the list. N is determined by converting the index to an integer value.

If the index is fractional, the fraction is truncated, not rounded. If the resulting integer is less than one or greater than the number of entries in the list, then no branch takes place and execution continues with the statement immediately following the branch instruction.

bump

label **`bump`** *charvar*
label **`bump`** *charvar prep numexp*

charvar is a destination character variable
numexp is the amount to bump the form pointer

Flags affected: eos

The bump statement increments or decrements the form pointer of the destination variable. The new form pointer value will be the value of the second operand plus the current value of the form pointer. If a second operand is not specified, the new form pointer value will be the current value of the form pointer plus one.

If the value of the second operand is fractional, the value is obtained by truncation, not rounding.

The resulting value of the form pointer must be between one and the logical length pointer value. If it is not, then no change is made to the form pointer and the eos flag is set.

call

label	**`call`**	*prog-label*
label	**`call`**	*prog-label prep list*
label	**`call`**	*prog-label* **`:`***objvar*
label	**`call`**	*prog-label* **`:`***objvar prep list*
label	**`call`**	*prog-label* **`if`** *cond*
label	**`call`**	*prog-label* **`if not`** *cond*
label	**`call`**	*prog-label* **`if`** *function-key*
label	**`call`**	*prog-label* **`if not`** *function-key*
label	**`call`**	*prog-label* **`if`** *expression*
label	**`call`**	*prog-label* **`if not`** *expression*

prog-label is a program execution label or a program label variable, except only a program execution label for class method calls
list is a list of variables, literals, and expressions
cond is one of **`equal`**, **`less`**, **`over`**, **`eos`** or **`greater`**
function-key is one of **`F1`**, **`F2`**, **`F3`**, **`F4`**, **`F5`**, **`F6`**, **`F7`**, **`F8`**, **`F9`**, **`F10`**, **`F11`**, **`F12`**, **`F13`**, **`F14`**, **`F15`**, **`F16`**, **`F17`**, **`F18`**,**`F19`**, **`F20`**, **`up`**, **`down`**, **`left`**, **`right`**, **`insert`**, **`delete`**, **`home`**, **`end`**, **`pgup`**, **`pgdn`**, **`tab`**, **`bktab`**, **`esc`**, or **`enter`**
expression is an algebraic expression

Flags affected: none

See also: keyin, moveadr, popreturn, pushreturn, setendkey, return, routine, verb

The call statement causes program execution to continue conditionally or unconditionally at the statement specified by *prog-label*. If *prog-label* is a program execution label, execution will continue at the statement with that label. If *prog-label* is a program label variable, execution will continue at the statement with the label whose value has been most recently assigned to the label variable.

The forms of the call statement without the **if** keyword cause program execution to continue at the statement specified by *prog-label*. If one of the other formats is used, program execution continues at the statement specified by *prog-label* only if the condition or expression following the **if** is true or the condition or expression following the **if not** is false.

In all cases, if execution will continue at *prog-label*, then the address of the statement following the call statement is put onto the top of the return stack for later use by the return statement. The return stack holds up to 200 return addresses.

The *function-key* forms of the call statement are only applicable following a keyin statement that was interrupted when a function key was pressed. Each function-key condition may only be tested once with a call, goto, or return. Conditions are reset after they are checked. All function-key conditions are also reset at the start of keyin execution.

The forms of the call statement that contain *prep list* are call with parameter statements. Any variable, literal or expression is allowed as a parameter. Label variables must be prefixed with the **~** character. The call with parameters statement is used in conjunction with the routine statement.

Call statements may call the same program label without an intervening return, but they are not recursive and expressions may not be used as parameters for such re-entrant executions of a call statement.

The forms of the call statement that contain **`:`***objvar* are object-oriented calls to class methods. Execution will continue at the program label with the name specified that is contained in the class for which the object variable was instantiated (with the make statement). If the object variable is not instantiated, an E 563 error occurs. If there is no method of that name in the appropriate class, then an E 551 error occurs.

ccall

label **`ccall`** *charexp*
label **`ccall`** *charexp prep list*

charexp is the source operand
list is a list of variables and literals

Flags affected: depends on ccall routine

See also: cverb

The ccall statement executes a user-written C language routine.

chain

label `chain` *charexp*

charexp is the source operand

Flags affected: none

See also: display, keyin, loadmod

The chain statement loads and begins execution of a new program (primary module). The logical string of the source operand is a file name of a compiled program. If no extension is supplied, **.dbc** is assumed. The chain statement unloads the previous primary module and all modules loaded with the loadmod statement. Certain keyin and display attributes are also reset.

change

label `change` *resource* `,` *charexp*
label `change` *device* `,` *charexp*
label `change` *resource* `,` *charexp; list*
label `change` *device* `,` *charexp; list*

resource is the resource variable
device is the device variable
charexp is the function operand
list is an optional list of character and numeric variables, literals, and control codes

Flags affected: none

See also: open, query

The change statement sends the logical string of the function operand and the optional list of variables, literals, and control codes to the specified resource or device.

***sl** is the blank suppression control code. This control code affects all remaining character variables in the list unless it is cancelled by another control code. This control code causes characters from the first character through the character pointed to by the length pointer to be sent, but no blanks are sent for the characters between the length pointer and the maximum length of the string. If the variable is cleared, then no characters are sent.

***ll** is the logical length control code. This control code is in effect for all character variables remaining in the list unless it is cancelled by another control code. The logical string of the character variable is sent.

***pl** is the physical length control code. This control code is in effect for all character variables remaining in the list unless it is cancelled by another control code. This control code causes characters from the first character through the character pointed to by the length pointer to be sent, followed by blanks for each of the characters between the length pointer and the maximum length of the string. If the variable is cleared, blanks are sent for all characters in the variable. This is the default mode for character variables.

charrestore

label **`charrestore`** *charexp*

`charrest` may be used in place of **`charrestore`**

charexp is the source operand

Flags affected: none

See also: charsave

The charrestore statement displays the logical string of the source operand in the current subwindow. The characters stored in the logical string of the source are displayed left-to-right, and top-to-bottom into the current subwindow. If there are not enough characters in the source to fill the current subwindow, then the remaining characters in the subwindow are not changed. If there are too many characters in the source to display, the extra characters are disregarded. The current video attributes apply to each character displayed. After the charrestore operation is executed, the cursor is positioned in the upper left corner of the subwindow.

charsave

label **`charsave`** *charvar*

charvar is the destination operand

Flags affected: eos

See also: charrestore

The characters displayed in the current subwindow are stored in the destination variable starting at the first character of the variable. The characters are saved in order from left-to-right and top-to-bottom. Each character in the window uses one character in the string. The destination form pointer is set to one and the length pointer is set to the number of characters stored.

If the destination variable is not big enough to store all the characters in the subwindow, then eos is set. Otherwise, eos is cleared.

check10

label **`check10`** *charvar1 prep charexp*
label **`check10`** *charvar1 prep charexp* **`,`** *charvar2*

`ck10` may be used in place of **`check10`**

charvar1 is the source character variable
charexp is the character variable, character literal, or character expression used as the weighting factor
charvar2 is the destination character variable

Flags affected: equal, over

The check10 statement verifies that the modulo 10 check digit is correct and optionally returns the check digit in the destination variable.

The logical strings of the source and weighting factor are used. The logical length of the source must be exactly one greater than the logical length of the weighting factor. If either of the first two operands is null or contains nondigits, or if the number of digits in the weighting factor is not one less than that of the source, over is set and no further action takes place. The last character of the source logical string is the check digit.

Assume M is the length of the source logical string and N is the length of the weighting logical string. Remember that N+1=M. The check digit is computed by taking each character of the source string and multiplying that by the respective character of the weighting string. This is done for N digits.

Each product is then converted to a number between 0 and 13 by adding the individual digits of the product. The resulting sums are then added. The new sum is divided by 10 and the remainder is subtracted from 10 to yield the check digit.

The resulting check digit value is then compared with the M^{th} digit of the source string. If they are the same, equal is set.

If the second format of the check10 statement is used, the result of the calculation is stored in the first character position of the destination variable. The form pointer and length pointer are set to one. If the check digit was not calculated because the source string or weighting string was invalid, a blank character is stored in the destination string.

check11

label **`check11`** *charvar1 prep charexp*
label **`check11`** *charvar1 prep charexp* `,` *charvar2*

`ck11` may be used in place of **`check11`**

charvar1 is the source character variable
charexp is the character variable, character literal, or character expression used as the weighting factor
charvar2 is the destination character variable

Flags affected: equal, over

The check11 statement verifies that the modulo 11 check digit is correct and optionally returns the check digit in the destination variable.

The logical strings of the source and weighting factor are used. The logical length of the source must be exactly one greater than the logical length of the weighting factor. If either of the first two operands is null or contains nondigits, or if the number of digits in the weighting factor is not one less than that of the source, over is set and no further action takes place. The last character of the source logical string is the check digit.

Assume M is the length of the source logical string and N is the length of the weighting logical string. Remember that N+1=M. The check digit is computed by taking each character of the source string and multiplying that by the respective character of the weighting string. This is done for N digits.

Each product is then converted to a number between 0 and 13 by adding the individual digits of the product. The resulting sums are then added. The new sum is divided by 11 and the remainder is subtracted from 11 to yield the check digit. If the check digit value is 10, the check digit is the letter A. If the check digit value is 11, the check digit is the letter B.

The resulting check digit is then compared with the M^{th} digit of the source string. If they are the same, equal is set. equal is set if the check digit is 10 or 11.

If the second format of the check11 statement is used, the result of the calculation is stored in the first character position of the destination variable. The form pointer and length pointer are set to one. If the check digit was not calculated because the source string or weighting string was invalid, a blank character is stored in the destination string. If the check digit value is a 10 or 11, a letter A or B is stored in the destination string, respectively.

chop

label **`chop`** *charexp prep charvar*

charexp is the source operand
charvar is the destination operand

Flags affected: eos

All characters in the logical string of the source operand are moved to the destination except for trailing blanks. The destination form pointer is set to one and the logical length pointer is equal to the number of characters moved. If the destination is not large enough to contain all characters to be moved, then as many characters that will fit are moved, and eos is set.

clear

label **`clear`** *list*

list is a comma delimited list of character and numeric variables, arrays, and list variables

Flags affected: none

See also: set

The clear statement zeroes and clears each element of the list. If an entry in the list is a numeric variable, it is set to zero. If an entry in the list is a character variable, its form pointer and length pointer are set to zero. If an entry in the list is a list variable, then each of the variables in the list is zeroed or cleared. If an entry in the list is an array name, then each element of the array is zeroed or cleared. If an entry is any other type of variable, the compiler will produce an error.

clearadr

label **`clearadr`** *list*

list is a comma delimited list of address variables and arrays of address variables

Flags affected: none

The clearadr statement clears each address variable in the list. If the list contains an array of address variables, then each element of the array is cleared.

clearendkey

label **`clearendkey`** *list*

list is a comma delimited list of numeric variables and arrays, decimal constants, and equate labels

Flags affected: none

See also: getendkey, setendkey

The clearendkey statement disables the keyin ending keys specified in the list. When an ending key is disabled, it is ignored. Each value in the list corresponds to an ending key. The list of keys is described in the Keyboard and Display Manipulation Statements section of the DB/C Programming Language General Information chapter. Negative values in the list are ignored.

The current ending keys are saved and restored by the statesave and staterestore statements, and by the scrnsave and scrnrestore statements.

On initial entry to the DB/C DX runtime the ending keys are set to **F1**, **F2**, **F3**, **F4**, **F5**, **F6**, **F7**, **F8**, **F9**, **F10**, **F11**, **F12**, **F13**, **F14**, **F15**, **F16**, **F17**, **F18**, **F19**, **F20**, **up**, **down**, **left**, **right** and **enter**. Additional ending keys will be active if **dbcdx.keyin.endkey=xkeys** runtime property is specified If the value 0 is contained in the list, all the ending keys are disabled with the exception of the **enter** key. The **enter** key may also be cleared by also specifying 256. Thus the statement, **clearendkey 0, 256** causes all ending keys to be disabled. Note that any subsequent non-timeout, non-keyin-continuous keyin statements will not terminate.

clearlabel

label **`clearlabel`** *lblvar*

lblvar is a label variable

Flags affected: none

See also: movelv, movevl, movelabel

The clearlabel statement clears the label variable.

clock

label **`clock`** *item prep charvar*

item is one of: **`time`**, **`day`**, **`year`**, **`date`**, **`calendar`**, **`utccalendar`**, **`timestamp`**, **`utctimestamp`**, **`utcoffset`**, **`weekday`**, **`version`**, **`release`**, **`pid`**, **`port`**, **`user`**, **`env`**, **`error`**, **`cmdline`**

charvar is the destination operand

Flags affected: eos

The clock statement provides operating system information. A string of characters is placed in the destination operand. If the destination is not large enough to contain the entire string, the eos flag is set. The value of the string of characters placed in the destination depends on the item.

`time`	moves the current time into the destination. The string format is: *hh*`:`*mm*`:`**`ss`**. The logical length is eight.
`day`	moves the day of the year into the destination. The format is: *ddd*. The logical length is three.
`year`	moves the year into the destination. The format is: *yy*. The logical length is two.
`date`	moves the date into the destination. The format is: *mmddyy*. The logical length is six.
`calendar`	moves a string that contains the current time and date into the destination. The format is: *Day Mth dd yyyy hh*`:`*mm*`:`*ss*. The logical length is 24. *Day* is the three-letter abbreviation for the day of the week. *Mth* is a three-letter abbreviation for the month.
`utccalendar`	moves a string that contains the UTC (GMT) time and date into the destination. The format is the same as for **`calendar`**.
`timestamp`	moves a string that contains the current time and date into the destination. The format is: *yyyymmddhhmmsspp*. The logical length is 16. *yyyymmdd* is the date. *hhmmsspp* is the time. In some runtime environments, *pp* is hundredths of a second. In other environments, *pp* is always **00**.
`utctimestamp`	moves a string that contains the current time and date into the destination. The format is the same as for **`timestamp`**.
`utcoffset`	moves a string that is the offset of this timezone from the prime meridian (GMT). The logical length is 5. The first character is **`+`** or **`-`**. The remaining characters are digits. For example, UTC offset for New York city when not on daylight savings time is **"-0500"**.
`weekday`	moves a single digit into the destination which represents the day of the week. The logical length is one. Sunday is represented by 1 and Saturday is represented by 7.
`version`	moves the DB/C DX version into the destination. The format of the resulting string depends on the operating system.
`release`	moves the string that contains the exact release number including minor revision numbers to the destination. The logical length varies, but the result will not contain any blanks.
`pid`	moves a string that is the operating system process id or process number of this instance of the DB/C DX runtime to the destination. The logical length varies.
`port`	moves a string that contains the port number into the destination. The logical length is three. The port number string is always zero filled on the left. This value can be provided by a runtime property.
`user`	moves a string that contains the user ID into the destination. This value can be provided by a runtime property.

env	moves a string into the destination that contains the environment information. The environment variables are separated with blanks.
error	moves a string that contains the most recent error message into the destination. If there has not been an error, or a previous clock error statement has retrieved the error, the form pointer and length pointer of the destination are set to zero.
cmdline	moves a string into the destination that contains the characters on the command line used to start the DB/C DX runtime. The contents of this string are operating system dependent.
ui	moves a three character string into the destination. If the GUI Smart Client is running then the returned value is **"GSC"**. If the character mode Smart Client is running then the returned value will be **"CSC"**. The value returned is **"GUI"** if Smart Client is not running and the GUI DB/C DX runtime is running. The value returned is **"CUI"** if Smart Client is not running and the character mode version of DB/C DX is running.

close, closeall

label `close` *file*
label `close` *file* `,` *mode*
label `close` *ifile*
label `close` *ifile* `,` *mode*
label `close` *afile*
label `close` *afile* `,` *mode*
label `close` *resource*
label `close` *device*
label `closeall`

file is the label of a file declaration
ifile is the label of an ifile declaration
afile is the label of an afile declaration
mode is the file close mode which is one of: `unchanged` or `delete`
resource is the resource variable
device is the device variable

Flags affected: none

See also: open, prepare

In the first two forms, the close statement logically disconnects a data file from the file variable in the DB/C program.

In the third and fourth forms, the close statement logically disconnects a data file and its index from the ifile variable in the DB/C program.

In the fifth and sixth forms, the close statement logically disconnects a data file and its associative index from the afile variable in the DB/C program.

If no file code mode is specified or if the **unchanged** file close mode is specified, then no other action is taken If the file, ifile, or afile variable was opened in exclusive mode and the **delete** file close mode is specified, then the close statement deletes the file. If the **delete** file close mode is specified but exclusive file open mode was not specified, then the file close mode is ignored. Note that for ifile and afile variables, both the data file and the index file will be deleted.

A close following immediately after a prepare statement for the same file variable will behave the same as if the **delete** file close mode was specified.

In the final two forms of close, the resource or device variable is logically disconnected from the resource or device.

In all cases, an attempt to close a file, resource, or device variable that is already closed is ignored.

The closeall statement closes all open files, print files, communications files, devices and resources.

cmatch

label	**`cmatch`**	*charvar prep charvar*
label	**`cmatch`**	*charvar prep charlit*
label	**`cmatch`**	*charvar prep hexcon*
label	**`cmatch`**	*charlit prep charvar*
label	**`cmatch`**	*charlit prep charlit*
label	**`cmatch`**	*charlit prep hexcon*
label	**`cmatch`**	*hexcon prep charvar*
label	**`cmatch`**	*hexcon prep charlit*
label	**`cmatch`**	*hexcon prep hexcon*

charvar is a character variable
charlit is a one-character literal
hexcon is a constant or equate label

Flags affected: equal, less, eos

The cmatch statement compares the value of one character from each operand. If an operand is a character variable, the form pointed character is used. If an operand is a decimal, octal, or hexadecimal constant, the character represented by that character code is used in the comparison.

If the characters compared are the same, equal is set and less is cleared. If the characters do not match, equal is cleared and less is set if the first operand character is greater. If the form pointer of either operand is zero, eos is set, equal and less are cleared, and no comparison is made.

cmove

label **`cmove`** *charvar1 prep charvar2*
label **`cmove`** *charlit prep charvar2*
label **`cmove`** *hexcon prep charvar2*

charvar1 is the source character variable
charlit is the source one-character literal
hexcon is the source constant or equate label
charvar2 is the destination character variable

Flags affected: eos

The cmove statement moves a single character from the source into the destination.

The source character is moved to the destination at the character position pointed to by the destination form pointer. If the source is a character variable, the character pointed to by the form pointer is moved to the destination. If the source is a decimal, octal, or hexadecimal constant, the character represented by that character code is moved to the destination. No changes are made to the form pointers or logical length pointers of either operand.

If the form pointer of either operand is zero, no move takes place and eos is set.

comclr

label **`comclr`** *cfile*

cfile is the label of a comfile declaration

Flags affected: less

See also: recv, recvclr, send, sendclr

The comclr statement sets the status of the communications file to "clear". If the prior status was "send-pending" or "receive-pending", then the send or recv operation is canceled and less is set.

comctl

label **`comctl`** *cfile*`,` *charvar*

cfile is the label of a comfile declaration
charvar is a character variable containing the communications control characters

Flags affected: none

The comctl statement is the communications control statement used to send and receive special control information.

comopen, comclose

label **`comopen`** *cfile*`,` *charexp*
label **`comclose`** *cfile*

cfile is the label of a comfile declaration
charexp is the variable, literal, or character expression that contains the resource name

Flags affected: none

See also: recv, recvclr, send, sendclr

The comopen statement opens a communications resource. The communications resource name in the second operand is operating system dependent. If the second operand is a character variable, the logical string is used. If the file does not exist or cannot be accessed because of security considerations, an error occurs. If a comopen statement is attempted on a comfile that is already open, then an implicit comclose is performed before the comopen.

The comclose statement logically disconnects a communications file from the communications resource.

compare

label **`compare`** *numexp1 prep numexp2*
label **`compare`** *numexp1 prep numarray2*
label **`compare`** *numarray1 prep numexp2*
label **`compare`** *numarray1 prep numarray2*

`comp` may be used in place of **`compare`**

numexp1 is the first source operand
numexp2 is the second source operand
numarray1 is the first source operand
numarray2 is the second source operand

Flags affected: equal, less, over

The compare statement compares numeric values. The flag settings indicate the result of the comparison.

The compare operation is similar to the subtract operation, except the result is not placed in a destination operand and overflow cannot occur. The first source operand is subtracted from the second and the result is examined. If the result is equal to zero, equal is set; otherwise it is cleared. If the result is less than zero, less is set; otherwise it is cleared. over is always cleared.

If either operand is an array, multiple comparisons take place. equal is set if all comparisons are equal. less is set if any comparison is less.

compareadr

label **`compareadr`** *var1 prep var2*

var1 is the first source operand
var2 is the second source operand

Flags affected: equal

The compareadr statement compares the address of the two variables. equal is set if the two variables contain the same address; otherwise equal is cleared.

comtst

label **`comtst`** *cfile*

cfile is the label of a comfile declaration

Flags affected: equal, over, less, eos

See also: comwait, recv, send, wait

The comtst statement is used to test the status of the comfile. This operation should be executed after a send, recv, comwait, or wait statement to test the status of the transmission.

The following table describes the flag settings:

Status	equal flag	eos flag	less flag	over flag
send clear	clear	clear	–	–
send pending	clear	set	–	–
send done - success	set	clear	–	–
send done - error	set	set	–	–
receive clear	–	–	clear	clear
receive pending	–	–	clear	set
receive done - success	–	–	set	clear
receive done - error	–	–	set	set

The comwait or comtst operation must be executed after a recv statement because these operations cause the data to be loaded into the list of variables. If a send or recv times out, the status will be set to send done - error or receive done - error, respectively.

comwait

label **`comwait`**
label **`comwait`** *cfile*

cfile is the label of a comfile declaration

Flags affected: none

See also: comtst, recv, recvclr, send, sendclr, wait

The comwait statement suspends program execution until a send or recv is satisfied, or until a send or recv timeout period elapses, or until an error occurs.

If the first format is used, then program execution is suspended until any currently open comfile has a status other than "clear" or "pending".

If the second format is used, then program execution is suspended until the specified comfile has a status other than "clear" or "pending".

console

label **`console`** *list*

list is a list of character and numeric variables, arrays, list variables and literals

Flags affected: none

The variables and literals in the list are written to the console device.

count

label **`count`** *numvar prep list*

numvar is the destination operand
list is a list of character and numeric variables, arrays, and list variables

Flags affected: equal, over

The number of keystrokes required to enter the variables in the list are counted and stored in the destination operand.

The number of keystrokes required to enter a character variable is the logical length of the variable less trailing blanks.

For a numeric variable, all digits plus the decimal point and minus sign are counted. Leading blanks and trailing zeros after the decimal point are not counted. The decimal point is not counted either if only zeros follow it.

If the count is zero, equal is set. If the count does not fit into the destination variable, over is set.

debug

label **`debug`**

Flags Affected: none

If the program is running in a debugger environment, the debug statement causes the DB/C DX runtime to suspend execution and enter the source debugger. If the program is not running in a debugger environment, this statement does nothing.

delete

label `delete` *file*
label `delete` *ifile, charvar*
label `delete` *afile*

ifile is the label of an ifile declaration
charvar is the source key
afile is the label of an afile declaration

Flags affected: over

See also: deletek

The delete statement is used to delete a record from a data file and a key from an index file. The delete operation for sequential or random files deletes the most recently accessed record from the data file. The delete operation for an indexed file deletes a key from the index that matches the source key. It also deletes from the data file the logical record that is associated with the key.

For an indexed file, the logical string from the character variable that is the source key is used to identify the logical record to be deleted. The source key matches the key in the index if they contain identical characters and have the same number of characters. The source key also matches the key in the index if the source key has more characters than the index key, all characters in the index key match characters in the source key, and all remaining characters in the source key are blank. If neither of these cases occurs, the source key does not match a key in the index.

If the source key matches a key in the index, then the record and the index key are deleted. If the source key does not match a key in the index, over is set and no change occurs. If the form pointer of the source key operand is zero, it is considered null, and the last accessed record is deleted. If the last access to the index file was unsuccessful, then an error occurs.

The delete operation for an AIM file deletes the most recently aimdexed accessed record from the data file.

deletek

label **`deletek`** *ifile*`,` *charvar*

ifile is the label of an ifile declaration
charvar is the source key

Flags affected: over

See also: delete

The deletek statement deletes a key from the index that matches the source key. No change is made to the logical records in the indexed file.

The logical string from the character variable that is the source key is used to identify the index key to be deleted. The source key matches the key in the index if they contain identical characters and have the same number of characters. The source key also matches the key in the index if the source key has more characters than the index key, all characters in the index key match characters in the source key, and all remaining characters in the source key are blank. If neither of these cases occurs, the source key does not match a key in the index.

If the source key matches a key in the index, then the index key is deleted. If the source key does not match a key in the index, over is set and no change occurs. If the form pointer of the source key operand is zero, it is considered null, and the last accessed key is deleted. If the last access to the index file was unsuccessful, then an error occurs.

destroy

label **`destroy`** *objvar*

objvar is the object variable

Flags affected: none

See also: call, make

The destroy statement causes the object referred to by the object variable to go out of existence. All variables that were created by the make statement are destroyed. If the object variable does not refer to an instantiated object, the destroy statement does nothing.

If a destroy routine exists for the class for which the object variable was instantiated, then the destroy routine, as well as any inherited destroy routines, are called before the variables go out of existence. The deepest ancestor destroy routine is called last and the rest of the destroy routines are called in reverse order of inheritance. If any of the destroy routines cannot be found, then an E 562 error occurs.

If the destroy statement is executed in the runtime scope of the instance of the class that the object variable refers to, then the instance goes out of existence and an implicit return is made to the statement following the method call that caused this instance to gain scope. The return stack is adjusted accordingly.

disable

label **`disable`**

Flags affected: over

See also: enable

The disable statement causes all keystroke and timeout trap events to be deferred until an enable statement executes, a keyin statement executes, or a program chaining occurs. Program chaining can occur as a result of a chain statement, a stop statement, or an error that is not trapped.

If the disable state is currently active, then this statement is ignored and the over flag is set. Otherwise, the over flag is cleared.

display

label **`display`** *list*

list is a comma delimited operand list of variables, literals, octal and hexadecimal constants, expressions, and display control codes

Flags affected: none

See also: format, keyin

The display statement displays characters on the screen, positions the cursor, alters display attributes, and affects subwindow dimensions. The operands identify the items to be displayed and their formats. The list of operands consists of character or numeric variables and arrays, literals, constants, expressions, and `*` control codes.

Normally, display advances the cursor to the first column of the line following the last character displayed. If the last character displayed is on the bottom line of the subwindow, all lines in the subwindow are scrolled up one line and the cursor is positioned in the lower left corner of the subwindow. If a semicolon (`;`) follows the last item in the operand list, the cursor will be positioned directly after the last character displayed. The use of the semicolon protects against inadvertent screen scrolling. A runtime property may modify the feature.

Control codes modify the manner in which data is displayed on the screen.

When a numeric variable is encountered in the list of operands, its characters are displayed starting at the current cursor position. The cursor remains on the position following the last character displayed.

When a character variable is encountered in the list of operands, the characters from the first physical character of the variable through the character pointed to by the logical length pointer are displayed. Blanks are displayed for the number of characters that are between the length pointer of the variable and the maximum length of the variable. In this way, the number of characters displayed is the maximum length of the character variable. If the form pointer of the variable is zero, blanks are displayed to the maximum length of the variable.

When a literal is encountered in the list of operands, its characters are displayed starting at the current cursor position.

When an octal or hexadecimal constant is encountered in the list of operands, the character represented by that character code is displayed in the same manner as a single character in a character variable.

When an expression is encountered in the list of operands, the expression is evaluated and the result is displayed in the same manner as a character or numeric variable.

`*`p=*h***`:`***v* is the cursor positioning control code. *h* and *v* may be decimal constants or numeric variables. The equal sign is optional. This control code resets the cursor position to the *h* horizontal position on the *v* line of the current subwindow.

`*`hu is the home up control code. This control code positions the cursor in the upper left corner of the current subwindow. It has the same effect as **`*`p=1:1**.

`*`hd is the home down control code. This control code positions the cursor in the bottom left corner of the current subwindow.

`*`eu is the end up control code. This control code positions the cursor in the upper right corner of the current subwindow.

`*`ed is the end down control code. This control code positions the cursor in the lower right corner of the current subwindow.

`*`v=*v* is the set vertical position control code. A colon (`:`) or a blank space may replace the equal sign. *v* is a decimal constant or numeric variable which represents the vertical position of the cursor in the current subwindow.

***va**=*n* is the adjust vertical position control code. A colon (**:**) or a blank space may replace the equal sign. *n* is a signed decimal constant or numeric variable which represents the number of vertical positions the cursor is to be moved relative to the current cursor position.

***h**=*h* is the set horizontal position control code. A colon (**:**) or a blank space may replace the equal sign. *h* is a decimal constant or numeric variable which represents the horizontal position of the cursor in the current subwindow.

***ha**=*n* is the adjust horizontal position control code. A colon (**:**) or a blank space may replace the equal sign. *n* is a signed decimal constant or numeric variable which represents the number of horizontal positions the cursor is to be moved relative to the current cursor position.

***n**=*n* or ***n** is the next line control code. *n* is an equate, decimal constant, or numeric variable that represents the number of lines the cursor will advance. The cursor position is set to horizontal position one on the new line. If the current cursor position is on the bottom line of the subwindow, the subwindow is scrolled up until the cursor has advanced the specified number of lines. If *n* is equal to zero, the control code is ignored. If *n* is not specified, the cursor is positioned on the first column of the next row (that is, *n* defaults to one). A runtime property may modify this feature.

***c** or ***c**=*n* is the carriage return control code. The cursor is positioned on the first column of the current line. *n* is an equate, decimal constant, or numeric variable. If *n* is equal to zero, the control code is ignored. Otherwise, *n* has no effect on the functioning of the control code.

***l** or ***l**=*n* is the line feed control code. This control code positions the cursor in the next row of the current column. If the current cursor position is on the last line of the subwindow, the screen is rolled up one line. *n* is an equate, decimal constant, or numeric variable that represents the number of lines the cursor will advance. If n is equal to zero, the control code is ignored. A runtime property may modify this feature.

***curson** is the cursor on control code. This code causes the cursor to appear on the screen during keyin of a variable. By default, the cursor will appear on the screen during keyin of a variable. Therefore, it is only necessary to specify the ***curson** control code if the ***cursoff** control code has been previously specified or if the cursor mode has been changed.

***cursoff** is the cursor off control code. This code suppresses the cursor display on the screen. This code is canceled by the ***curson** code, by chaining to another program, or by changing the cursor mode.

***cursor**=*mode* is the control code that controls the cursor mode. mode may be one of ***on**, ***off**, or ***norm**.

***cursor=*on** means that the cursor will always be displayed. ***cursor=*off** means that the cursor will never be displayed. ***cursoff** is a synonym for ***cursor=*off**. ***cursor=*norm** means that the cursor will be off except when the program is in a keyin statement. ***curson** is a synonym for ***cursor=*norm**. ***cursor=*norm** is the default. The cursor mode will return to ***norm** upon execution of a chain statement.

***cursor**=*shape* is the control code that controls the cursor shape. *shape* may be one of ***uline**, ***half**, or ***block**. ***cursor=*uline** means that the cursor will be displayed as an underline. ***cursor=*half** means that the cursor will be displayed at half of its normal height. ***cursor=*block** means that the cursor will be displayed as a full block at its normal height. ***cursor=*block** is the default. The specified cursor shape does not affect the cursor mode. The cursor shape will return to ***block** upon execution of a chain statement.

***r** or ***r**=*n* is the screen roll up control code. The screen is rolled up and the cursor is left at the same location. *n* is an equate, decimal constant, or numeric variable that represents the number of lines the screen is rolled up. If *n* is equal to zero, the control code is ignored.

***rd** or ***rd**=*n* is the screen roll down control code. The screen is rolled down and the cursor is left at the same location. *n* is an equate, decimal constant, or numeric variable that represents the number of lines the screen is rolled down. If *n* is equal to zero, the control code is ignored.

***scrright**=*scroll-data* is the scroll right control code. A colon (**:**) or a blank space may replace the equal sign. *scroll-data* is a colon delimited list which includes: character variables and literals, numeric variables and literals, unreferenced arrays, list variables, decimal constants, octal constants, hexadecimal constants, equates, any of the display attributes control codes, or any of the graphics control codes (excluding the double line graphics control codes). In addition, the scroll-data list may include ***scrend**, which designates the end of the list. ***scrend** is not required. A colon can be used to continue the *scroll-dat*a list on the next program line. ***scrright** scrolls the current subwindow to the right by one column. *scroll-data* list items are placed top to bottom in the far left column of the current subwindow. Characters in the far right column are lost.

***scrleft**=*scroll-data* is the scroll left control code. A colon (**:**) or a blank space may replace the equal sign. *scroll-data* is a colon delimited list which includes: character variables and literals, numeric variables and literals, unreferenced arrays, list variables, decimal constants, octal constants, hexadecimal constants, equates, any of the display attributes control codes, or any of the graphics control codes (excluding the double line graphics control codes). In addition, the scroll-data list may include ***scrend**, which designates the end of the list. ***scrend** is not required. A colon can be used to continue the *scroll-data* list on the next program line. ***scrleft** scrolls the current subwindow to the left by one column. *scroll-data* list items are placed top to bottom in the far right column of the current subwindow. Characters in the far left column are lost.

***opnlin** is the open line control code. This code causes all lines in the current subwindow below the current line to be moved down one line. The characters in the current line from the current cursor position through the end of the line remain in the same columns but are shifted down one line. Data on the bottom line of the screen is lost.

***clslin** is the close line control code. This code causes the characters from the current cursor position through the end of the line to be erased. The erased characters are replaced by the characters in the same column positions in the line following the current line. The line following the current line is erased and a roll up occurs between this line and the bottom line of the current subwindow.

***inslin** is the insert line control code. This code causes a roll down to occur from the line containing the current cursor position through the bottom line of the current subwindow. A line of blank spaces is inserted in the line containing the current cursor position. The bottom line of the current subwindow is lost.

***dellin** is the delete line control code. This code causes the line containing the current cursor position to be erased. A roll up occurs from the line following the current line through the bottom line of the current subwindow. A line of blank spaces is inserted in the bottom line of the current subwindow.

inschr**=*h:***v* is the insert character control code. A colon (**:**) or a blank space may replace the equal sign. *h* and *v* are decimal constants or numeric variables are decimal constants or numeric variables which represent horizontal and vertical coordinates. All characters between the current cursor position and *h***:***v* are shifted to the right one space. One blank character is placed at the current cursor position. The character originally at *h***:***v* is lost.

delchr**=*h:***v* is the delete character control code. A colon (**:**) or a blank space may replace the equal sign. *h* and *v* are decimal constants or numeric variables which represent horizontal and vertical coordinates. All characters between the current cursor position and *h***:***v* are shifted to the left one space. The character originally at the current cursor position is lost. One blank character is placed at *h***:***v*.

***es** erases the entire subwindow. The cursor position is set to 1, 1.

***el** erases everything from the current cursor position through the end of the line.

***ef** erases all characters from the cursor through the end of the current line and also erases all subsequent lines in the subwindow.

rptchar**=*charexp:***n* is the repeat character control code. A colon (**:**) or a blank space may replace the equal sign. *charexp* is a character variable, character literal, character expression, or a graphics character control code. If *charexp* is a variable, then the form pointed character is the character to be repeated. *n* is a decimal constant or numeric variable that specifies the number of times the character is to be repeated. Line wrap will not take place.

rptdown=charexp***:***n* is the repeat character down control code. A colon (**:**) or a blank space may replace the equal sign. charexp is a character variable, character literal, character expression, or a graphics character control code. If *charexp* is a variable, then the form pointed character is the character to be repeated. *n* is a decimal constant or numeric variable that specifies the number of times the character is to be repeated. Characters are displayed from top to bottom, one per line.

***click** is the control code that causes a click to sound. This option is unavailable on some terminals.

***b** is the beep control code. This code causes a beep sound to occur.

***zs** is the zero suppress control code. This control code affects only the next variable in the list. If the next variable is a numeric variable with a zero value, blanks are displayed for each character position of the numeric variable (including the decimal point).

***zf** is the zero fill control code. This control affects only the next variable in the list. If the next variable is a numeric variable, then any blanks are replaced by zeros. In addition, the minus sign (if it exists) is displayed as the first character in the variable.

***sl** (or ***+**) in a display operation is the trailing blank suppression control code. This control code affects display of all character variables that follow in the list of operands. This control code displays character variables from the first physical character in the variable through the character pointed to by the length pointer. No blanks are displayed after the length pointer. If the form pointer of the variable is zero, no characters are displayed and the cursor position is not changed.

***ll** in a display operation is the logical string display control code. This control code affects display of all character variables that follow in the list of operands. This control code displays the characters contained in the logical string of the variable. If the form pointer of the variable is zero, no characters are displayed and the cursor position is not changed.

***pl** (or ***–**) in a display operation is the display suppression off control code. This control code causes display of character variables to revert to the normal method of display. It cancels the effects of the ***sl** and ***ll** control codes.

***dcon** in a display operation is the display comma control code. This control code causes a comma to be displayed instead of the decimal point (period) in numeric variables. This control code affects all numeric variables that follow. It is cancelled by the ***dcoff** control code or by chaining to another program. This control code also affects the operation of the keyin statement.

***dcoff** is the cancel display comma control code. This control code cancels the effect of the ***dcon** control code.

format=charexp* is the format control is the format control code. This control code affects the next variable in the list. *charexp* is a format mask which is used to reformat numeric and character data. Refer to the explanation of the format statement for information about how the mask is used.

***revon** (or ***hon**) is the video control code for reverse video display. This control code affects display of all variables and literals that follow in the list of operands. The control code is canceled by the ***revoff** and ***alloff** control codes, or by chaining to another program. This control code displays all characters in inverse video mode on the screen. If colors are in use, this control code reverses the back ground and foreground colors. This feature is unavailable on certain terminals.

***revoff** is the reverse video off control code. This control code cancels the effect of ***revon**. Characters are displayed in normal white on black, or foreground on background if colors are in use.

***boldon** (or ***dion** or ***v2lon**) is the video control code for high-intensity display. This control code affects display of all variables and literals that follow. It is canceled by the ***boldoff** and ***alloff** control codes, or by chaining to another program. All characters are displayed in high-intensity mode on the screen. This feature is unavailable on certain terminals.

***boldoff** (or ***dioff**) is the video control code that reverses the effect of the ***boldon** control code. This control code causes characters to be displayed in normal-intensity mode.

***ulon** is the video underline control code. This control code affects display of all variables and literals that follow in the operand list. It is canceled by the ***uloff** and ***alloff** control codes, or by chaining to another program. All characters are displayed with an underline. This feature is unavailable on certain terminals.

***uloff** is the video underline off control code. This control code cancels the effect of the ***ulon** control code.

***blinkon** is the video blink control code. This control code affects display of all variables and literals that follow in the operand list. It is canceled by the ***blinkoff** and ***alloff** control codes, or by chaining to another program. All characters that are displayed will blink. This feature is unavailable on certain terminals.

***blinkoff** is the video blink off control code. This control code cancels the effect of the ***blinkon** control code.

***black**, ***blue**, ***green**, ***cyan**, ***red**, ***magenta**, ***yellow**, and ***white** are the fixed color control codes. These codes affect display of all variables and literals that follow in the operand list. They are canceled by the ***coloroff** control code. All characters that are displayed will appear in the designated colors.

color=color* (or ***fgcolor=***color*) is the variable color control code. color may be a decimal constant, a numeric variable, or one of the fixed color control codes. If color is a decimal constant or a numeric variable, the values 0 through 7 correspond to the fixed color control codes respectively. The values 8 through 15 correspond to the highlighted fixed color control codes respectively. This control code affects display of all variables and literals that follow in the operand list. All characters that are displayed will be displayed in the color specified by color. This control code is canceled by ***coloroff**.

bgcolor=color* is the background color control code. color may be a decimal constant, a numeric variable, or one of the fixed color control codes. This control code affects display of all variables and literals that follow. All characters that are displayed will be displayed with the background color specified by color. This control code is canceled by ***coloroff**.

***coloroff** is the color off control code. This control code resets colors to the default colors (usually white on black).

***alloff** (or ***hoff**) is the video attributes off control code. The effects of all video attributes control codes are turned off. This control code has the effect of ***revoff**, ***boldoff**, ***uloff**, and ***blinkoff**. A DB/C DX runtime property may modify this feature.

***hln**, ***vln**, ***crs**, ***ulc**, ***urc**, ***llc**, ***lrc**, ***rtk**, ***dtk**, ***ltk**, ***utk**, ***upa**, ***dna**, ***lfa**, and ***rta** are the horizontal line, vertical line, crossed lines, upper left corner, upper right corner, right tick mark, down tick mark, left tick mark, up tick mark, up arrow, down arrow, left arrow, and right arrow graphics control codes, respectively. These control codes cause the specified graphics character to be displayed.

***dblon**, ***dbloff**, ***hdblon**, ***hdbloff**, ***vdblon**, and ***vdbloff** are the double line graphics control codes. These control codes affect the characters displayed by the graphics control codes, except the arrow graphics characters. ***dblon** causes all lines (both horizontal and vertical) to be displayed as double lines. ***dbloff** causes all lines to be displayed as single lines. ***hdblon** causes all horizontal lines to be displayed as double lines. ***hdbloff** causes all horizontal lines to be displayed as single lines. ***hdblon** and ***hdbloff** do not affect the display of vertical lines. ***vdblon** causes all vertical lines to be displayed as double lines. ***vdbloff** causes all vertical lines to be displayed as single lines. ***vdblon** and ***vdbloff** do not affect the display of horizontal lines.

setswtb=t***:***b* is the set window top/bottom control code. *t* and *b* are decimal constants or numeric variables that define the top and bottom vertical parameters of a subwindow. When a subwindow is defined, all control codes operate relative to the defined subwindow. The cursor position initially will be set to ***p=1:1**. The ***setswtb** control code is canceled by the ***resetsw** control code.

setswlr=l***:***r* is the set window left/right control code. *l* and *r* are decimal constants or numeric variables that define the left and right horizontal parameters of a subwindow. When a subwindow is defined, all control codes operate relative to the defined subwindow. The cursor position initially will be set to ***p=1:1**. The ***setswlr** control code is canceled by the ***resetsw** control code.

setswall=t***:***b***:***l***:***r* is the set window all control code. *t*, *b*, *l*, and *r* are decimal constants or numeric variables that define the vertical and horizontal parameters of a subwindow. When a subwindow is defined, all control codes operate relative to the defined subwindow. The cursor position initially will be set to ***p=1:1**. The ***setswall** control code is canceled by the ***resetsw** control code.

***resetsw** cancels the effects of the ***setswtb**, ***setswlr**, and ***setswall** control codes. The cursor position initially will be set to ***p=1:1**.

***raw** is the raw output control code. The effect of this control code is to ignore any window boundaries. This is useful when escape sequences are being displayed. This control code is canceled by ***rawoff** or by chaining to another program.

***rawoff** is the control code that cancels the effect of the ***raw** control code.

w=n*, ***w***n*, and ***w** cause the program to wait. *n* is an integer decimal constant that specifies the number of seconds for the program to wait. If *n* is not specified, the program waits one second.

***pon** is the printer on control code. The option is only available on certain terminals. The auxiliary printer port of the terminal is turned on by this control code. This control code also causes raw mode to be in effect. The ***pon** control code is canceled by the ***poff** control code or by chaining to another program.

***poff** is the control code that cancels the effect of the ***pon** control code.

divide

label **`divide`** *numexp1 prep numvar3*
label **`divide`** *numexp1 prep numexp2* **`giving`** *numvar3*
label **`divide`** *numexp1 prep numarray3*
label **`divide`** *numarray1 prep numvar3*
label **`divide`** *numarray1 prep numarray3*
label **`divide`** *numarray1 prep numarray2* **`giving`** *numarray3*

`div` may be used in place of **`divide`**

numexp1 is the first source operand
numexp2 is the second source operand
numvar3 is the destination operand
numarray1 is the first source operand
numarray2 is the second source operand
numarray3 is the destination operand

Flags affected: equal, less, over

If the first format is used, then the destination operand is divided by the source operand and the result is placed in the destination operand.

If the second format is used, then the second source operand is divided by the first source operand and the result is placed in the destination operand.

If the third format is used, the source operand is divided by each element of the destination operand.

If the fourth format is used, each element of the source array is divided successively into the destination operand. The result is placed in the destination operand.

If the fifth format is used, each element of the source array is divided by the corresponding element in the destination array.

If the sixth format is used, each element of the first source array is divided by the corresponding element of the second source array, and the result is placed in the corresponding element of the destination array.

The precision used for the intermediate result depends on the number of operands. In the following, **L** is the number of digits to the left of the decimal point in the intermediate result and **R** is the number of digits to the right of the decimal point in the intermediate result. **L1**, **R1**, **L2**, **R2**, **L3** and **R3** correspond with the digits in the first, second and third parameters, respectively.

In the two operand forms of the divide statement, the precision of the intermediate result is:

L = L2 + R1
R = R1 + R2

In the three operand forms of the divide statement, the precision of the intermediate result is:

L = L2 + R1
R = R3 + 1

Rounding takes place when the intermediate result is moved to the destination.

draw

label **`draw`** *image*`;`*list*

image is the destination image variable
list is a comma delimited list of parameters

Flags affected: none

The draw statement draws graphics and text on the destination image variable.

In non-graphical versions of DB/C DX, execution of the draw statement results in an error.

There is a current draw position that is always valid. This draw position is made up of a horizontal and a vertical component. The upper left corner of the screen is draw position 1,1. The draw position increases down and to the right. The units of the draw position are screen pixels.

The following keywords and values make up the list of parameters.

p=*h***:***v* is the set draw position parameter. *h* and *v* may be decimal constants or numeric variables. The current draw position is set to the *h* horizontal position and to the *v* vertical position.

h=*h* is the set horizontal draw position parameter. *h* may be a decimal constant or numeric variable. The horizontal component of the draw position is set to *h*. The vertical component is not changed.

v=*v* is the set vertical draw position parameter. *v* may be a decimal constant or numeric variable. The vertical component of the draw position is set to *v*. The horizontal component is not changed.

ha=*n* is the adjust horizontal draw position parameter. *n* may be a signed decimal constant or a numeric variable. The value of *n* is added to the current horizontal draw position. The vertical component is not changed.

va=*n* is the adjust vertical draw position parameter. n may be a signed decimal constant or a numeric variable. The value of *n* is added to the current vertical draw position. The horizontal component is not changed.

color=*color* is the set current draw color parameter. *color* may be a decimal constant, a numeric variable, or a fixed color code. If color is a decimal constant or numeric variable, it specifies the new color. This value is an integer that corresponds to a 24-bit RGB value (red is low eight bits, green is middle eight bits, and blue is high eight bits). Fixed color codes are: ***black**, ***blue**, ***green**, ***cyan**, ***red**, ***magenta**, ***yellow**, and ***white**.

erase is the erase screen parameter. The entire image is changed to the current draw color and the draw position is set to 1,1.

replace=*oldcolor***:***newcolor* is the replace color parameter. *oldcolor* and *newcolor* may be decimal constants, numberic variables, or fixed color codes just as in the **color=** parameter. All pixels in the image that are the color specified by *oldcolor* are changed to be the color specified by *newcolor*.

dot is the draw dot parameter. One pixel is drawn in the current draw color at the current draw position.

bigdot=*n* is the draw big dot parameter. *n* may be a decimal constant or numeric variable. A solid circle is drawn in the current draw color with the center at the current draw position and a radius of *n* pixels.

linewidth=*n* is the set line width parameter. *n* may be a decimal constant or numeric variable. The current line width is set to be *n* pixels wide. The default line width is one.

linetype=*linecode* is the set line type parameter. **linecode** may be ***solid** or ***revdot**. ***solid** means a solid line. ***revdot** means an exclusive-or dotted line. ***solid** is the default line type.

line=*h***:***v* is the draw line parameter. *h* and *v* may be decimal constants or numeric variables. A line is drawn from the current draw position to the position specified by *h* and *v*. The line is drawn in the current draw color of the current line width and type. The current draw position is reset to *h***:***v*.

circle=*n* is the draw circle parameter. *n* may be a decimal constant or numeric variable. A circle is drawn in the current draw color with the center at the current draw position and a radius of *n* pixels. The circle is drawn with a line that is the current line width and type.

box=*h* **:** *v* is the draw box parameter. *h* and *v* may be decimal constants or numeric variables. The current draw position is one corner of the box. *h* **:** *v* specifies the diagonally opposite corner of the box. The box is drawn with a line that is the current line width and type.

rectangle=*h* **:** *v* is the draw filled rectangle parameter. *h* and *v* may be decimal constants or numeric variables. The current draw position is one corner of the rectangle. *h* **:** *v* specifies the diagonally opposite corner of the rectangle. The filled rectangle is drawn in the current draw color.

triangle=*h1* **:** *v1* **:** *h2* **:** *v2* is the draw filled triangle parameter. *h1*, *v1*, *h2*, and *v2* may be decimal constants or numeric variables. The current draw position is one corner of the triangle. *h1* **:** *v1* is the second corner of the triangle. *h2* **:** *v2* is the third corner of the triangle. The filled triangle is drawn in the current draw color.

font=*font* is the set draw font parameter. *font* is a character variable or literal that specifies a font name.

linefeed is the draw text linefeed parameter. The vertical component of the draw position is increased by the height (in pixels) of the current font. The height includes normal interline space.

newline is the draw text new line parameter. The horizontal component of the draw position is set to one. The vertical component of the draw position is in creased by the height (in pixels) of the current font. The height includes normal interline space.

text=*text* is the draw text parameter. *text* is a character variable, numeric variable, or character literal that contains the string of characters to be drawn. The characters are drawn in the current draw color with the current font starting at the current draw position. The upper left corner of a character is its draw position. As each character is drawn the horizontal component of the draw position is increased incrementally by the width of the character drawn. The draw position is placed at the upper left corner of the character to the right of the last character drawn.

atext=*text* **:** *n* is the draw angled text parameter. *text* is a character variable, numeric variable, or character literal that contains the string of characters to be drawn. The characters are drawn such that the upper left corner of the text is at the current draw position. *n* defines the angle, in degrees, at which the text is drawn. The angle is the same as for a compass, that is north (up) is zero degrees, east (normal) is 90 degrees, south (down) is 180 degrees, and west (upside down) is 270 degrees. The draw position is not changed.

ctext=*text* **:** *n* is the draw centered text parameter. *text* is a character variable, numeric variable, or character literal that contains the string of characters to be drawn. The characters are drawn in the current draw color with the current font. The characters are drawn centered in the area bounded on the left by the current draw position and bounded on the right by the position defined by adding the current horizontal position with *n*. The draw position is not changed.

rtext=*text* is the draw right justified text parameter. *text* is a character variable, numeric variable, or character literal that contains the string of characters to be drawn. The characters are drawn such that the rightmost edge of the rightmost character is at the current draw position. The draw position is not changed.

image=*image* is the draw image parameter. *image* is an image variable. The image referred to by *image* is copied over a rectangle in the destination image. The current draw position defines the upper left corner of the rectangle.

clipimage=*image* **:** *h1* **:** *v1* **:** *h2* **:** *v2* is the clipped image copy parameter. image is an image variable. *h1*, *v1*, *h2*, and *v2* may be decimal constants or numeric variables. A portion of the image referred to by image is copied over the destination image. The portion of the image to copy is bounded by *h1* **:** *v1* and *h2* **:** *v2*.

stretchimage=*image* **:** *h* **:** *v* is the stretch/compress image copy parameter. image is an image variable. The image referred to by image is first stretched and/or compressed to be the size specified by *h* and *v*. This stretched/compressed image is then copied to a rectangle in the destination image. The current draw position defines the upper left corner of the rectangle.

imagerotate=*n* is the draw rotated image parameter. The image is drawn so that the upper left corner of the image is at the current draw position. *n* defines the angle, in degrees, at which the image is drawn. The angle is the same as for a compass, that north (up) is zero degrees, east (normal) is 90 degrees, south (down) is 180 degrees, and west (upside down) is 270 degrees. The draw position is not changed.

edit

label	**`edit`**	*var prep charvar1*
label	**`edit`**	*var prep charvar2, charvar3*
label	**`edit`**	*var prep charlit* `,` *charvar3*
label	**`edit`**	*var prep charvar2* **`giving`** *charvar3*
label	**`edit`**	*var prep charlit* **`giving`** *charvar3*

var is the source operand
charvar1 is the destination operand which contains mask characters
charvar2 is the operand which contains mask characters
charvar3 is the destination operand
charlit is the operand which contains mask characters

Flags affected: less, over, eos

The edit statement formats numeric and character data. It uses an editing mask which specifies the format of the result. The mask controls a character-by-character move from the source operand into the destination operand.

The data in the source operand is edited into the logical string of the destination operand. Editing takes place from left to right. Character by character, the logical string of the source is moved to the destination following the rules of the edit mask. The contents of the source operand are not changed. The destination form pointer and length pointer also are not changed.

The second operand contains special mask characters that denote how data from the source is moved to the destination. The mask can contain any character, but only the special characters affect the movement of data. Editing proceeds as follows: each character in the mask is compared with the special characters. If the character in the mask is not one of the special characters, then it is left unchanged and editing continues with the next character in the mask.

If the first form is used, the destination initially contains the editing mask. The logical string of the source is edited into the logical string of the destination.

If any of the remaining forms are used, the logical string of the source is edited into the destination operand, using the second operand as the editing mask. The contents of the second operand are not changed. The logical string of the second operand is moved to the third operand (the destination).

If the form pointer of the source or destination operand is zero, the eos flag is set and no editing takes place.

If the logical string of the destination is shorter than the logical string of the source, the eos flag is set and only those characters fit in the destination are moved. If the logical string of the destination is longer than the logical string of the source, the source is logically extended with blanks if it is a character variable or with zeros if it is a numeric variable.

If any edit error is detected while moving characters to the destination, the over flag is set. If no edit errors are detected while moving characters to the destination, the over flag is cleared. The less flag is set if a dollar sign character is stored in the destination over a non-zero digit.

The editing of a character source uses different mask characters than the editing of a numeric source.

Four special mask characters are applicable if the source operand is a character variable:

9	A character is moved from the source. An edit error occurs if the character moved is not a digit.
A	A character is moved from the source. An edit error occurs if the character moved is not an alphabetic or blank character.
B	The letter **B** in the destination is changed to a blank and no characters are moved from the source.
X	A character is moved from the source.

Nine special mask characters are applicable if the source operand is a numeric variable:

9 A character is moved from the source. An edit error occurs if the character moved is not a digit.

B The letter **B** in the destination is changed to a blank and no characters are used from the source.

Z Zero suppression is in effect for the character to be moved. If the character to be moved from the source operand is a leading zero, a blank is stored in the destination and the source character is discarded. Otherwise, the source character is stored in the destination and zero suppression is inactivated.

, The comma remains in the destination and no character is moved from the source. If zero suppression is in effect, the comma is changed to a blank character.

. The period is left unchanged in the destination. Zero suppression is in activated. If a floating sign was requested and has not been stored in the destination, it is stored in the character to the left of the period.

+ This character must appear as the first or last character in the logical string of the destination. If it is the first character in the logical string, a **+** or **–** sign will be stored in the first position. If it is the last character in the logical string, then a **+** or **–** sign will be stored in the last character position.

– This works the same as a **+** sign except that if the source is positive, a blank is stored instead of a **+** sign.

$ Dollar sign fill is in effect for the character to be moved. If the character to be moved is **0** and no non-zero digits have been stored in the destination, then the dollar sign character is left unchanged and the character from the source is discarded. At least one dollar sign character will always be stored in the destination, even if a significant digit has to be discarded.

***** Asterisk fill is in effect for the character to be moved. If the character to be moved is a leading zero, the asterisk character remains in the destination and the character from the source is discarded.

empty

label **`empty`** *list*

list is a comma delimited list of queue variables

Flags affected: none

The empty statement discards all messages in the specified queue(s).

enable

label `enable`

Flags affected: none

See also: disable

The enable statement causes all keystroke trap events to be enabled. This reverses the action of the disable statement.

endset

label **`endset`** *charvar*

charvar is the destination operand

Flags affected: none

See also: lenset

The form pointer of the destination is set to the same value as the length pointer.

erase

label `erase` *charexp*

charexp is the file name

Flags affected: over

The erase statement deletes the file specified by *charexp*. over is set if the specified file cannot be found. If the file exists and the erase statement is unsuccessful, an IO error occurs.

execute

label **`execute`** *charexp*

charexp is the source operand

Flags affected: over

The execute statement causes the command line in the logical string of the source operand to be executed by the operating system command interpreter (shell). A new task is created to execute that command line. Program execution continues immediately without waiting for the new task to finish.

If the source operand is a null string, over is set.

extend

label **`extend`** *charvar*
label **`extend`** *charvar prep dcon*
label **`extend`** *charvar prep numvar*

charvar is the destination operand
dcon is the source operand
numvar is the source operand

Flags affected: eos

The extend statement appends one or more blank spaces to the destination variable following the form pointed character.

If the first format is used, one blank is appended. The second and third formats define the number of blanks to be appended. If the numeric variable is fractional, the value is obtained by truncation, not rounding. If the number is less than one, no blanks are appended.

Only the number of blanks that will fit in the variable are appended. eos is set if fewer blanks are appended than were specified. The form pointer and length pointer are set to point to the last blank moved. If no blanks are appended, then the form pointer and length pointer are not changed.

external

label **`external`**

label is required

Flags affected: none

The external statement defines label as an external reference. In other words, label may be found in another module at execution time. The external statement must precede any references to its label unless the **-x** parameter is specified on the compiler command line.

File Manipulation Statements

label **`aimdex`** *charexp*
label **`build`** *charexp*
label **`copy`** *charexp*
label **`create`** *charexp*
label **`encode`** *charexp*
label **`exist`** *charexp*
label **`index`** *charexp*
label **`reformat`** *charexp*
label **`sort`** *charexp*

charexp is the source operand

Flags affected: over

Each statement causes the respective DB/C DX utility to be executed with the parameters specified in the source operand. For information about the DB/C DX utilities, refer to the DB/C DX Utilities chapter.

The over flag is cleared if the operation is successful. The over flag is set if the input file does not exist or if the command could not complete normally.

filepi

label **`filepi`** *dcon* `;` *list*
label **`filepi`** **`0`**

dcon is a decimal number
list is a list of file, ifile and afile variables

Flags affected: none

The filepi statement limits access to certain data files to one DB/C DX runtime at a time.

The first form of the filepi statement specifies the duration of the lock and a list of files to be locked. The first operand, *dcon*, specifies the number of instructions during which the files are to remain locked. The value of this decimal number may be 2 through 254, inclusive. If the value of the decimal number is 1, the statement is ignored. The second operand, *list*, specifies the files to be locked as a list of file, ifile and afile variables.

If the decimal number specified is non-zero, then other programs are not permitted to execute a filepi on the specified files. The lock is done on the logical name of the text file associated with the file, ifile or afile variable. Therefore, each of the file, ifile and afile variables must be open.

For example, assume a program is currently executing a filepi statement or any of the following statements. The filepi statement has locked all other programs out of a certain file. If another program requires access to the locked file, then execution of this second program is on hold until the effect of the filepi has terminated.

The following statements do not count as executing instructions for the filepi locking mechanism: goto, branch, if, else, endif, for, loop, while, until, repeat, call, perform, and return.

The following statements immediately terminate the effect of a filepi statement: keyin, display, beep, pause, splopen, splclose, print, release, chain, shutdown, stop, rollout, sqlcode, sqlexec, sqlmsg, wait, comwait and all other the File Manipulation Statements.

An error results when an attempt is made to lock files with a filepi operation while another filepi is still in effect.

The second form of the filepi statement is used to cancel all existing file locks. A filepi 0 statement can be used to unlock the files before the number of instructions specified in the initial filepi statement have been executed.

fill

label **`fill`** *charvar prep list*
label **`fill`** *charlit prep list*

charvar is the source character variable
charlit is the source one-character literal
list is the list of character variables and character array variables

Flags affected: none

The form pointed character of the source or the source one-character literal is moved to all character positions in the destination. The form pointer of the destination is set to one and the length pointer is set to the maximum length. If the source form pointer is zero, no changes are made.

flagrestore

label `flagrestore` *charexp*

`flagrest` may be used in place of `flagrestore`

charexp is the source operand

Flags affected: eos, equal, less, over

See also: flagsave

The flagrestore statement restores flag settings from a character variable. The logical string of the source operand is used to set or clear eos, equal, less, and over.

If the first character in the source operand is a one, then eos is set. If it is a zero, then eos is cleared.

If the second character in the source operand is a one, then equal is set. If it is a zero, then equal is cleared.

If the third character in the source operand is a one, then less is set. If it is a zero, then less is cleared.

If the fourth character is a one, then over is set. If it is a zero, then over is cleared.

If any of the characters has a value other than one or zero, then no change is made to the corresponding flag. If the source operand is less than four characters long, then only those flags that correspond to the existing characters are changed.

flagsave

label **`flagsave`** *charvar*

charvar is the destination operand

Flags affected: none

See also: flagrestore

The flagsave statement moves the values of eos, equal, less, and over into the destination operand. The flagsave statement does not affect the flag values; it merely saves them to a variable.

If the eos flag is set, then **1** is moved to the first position of the destination operand. If it is cleared, then **0** is moved.

If the equal flag is set, then **1** is moved to the second position of the destination operand. If it is cleared, then **0** is moved.

If the less flag is set, then **1** is moved to the third position of the destination operand. If it is cleared, then **0** is moved.

If the over flag is set, then **1** is moved to the fourth position of the destination operand. If it is cleared, then **0** is moved.

If the maximum length of the destination operand is less than four, then only those flag settings that fit are saved. The form pointer is set to one and the logical length pointer is set to the number of characters moved.

flusheof

label **`flusheof`** *file*
label **`flusheof`** *ifile*
label **`flusheof`** *afile*

`flush` may be used in place of **`flusheof`**

file is the label of a file declaration
ifile is the label of an ifile declaration
afile is the label of an afile declaration

Flags affected: none

See also: close, open, prepare

The flusheof statement causes the DB/C DX runtime to flush any data written by the program to the disk. In addition, the flusheof statement causes the operating system to flush it's own buffers for the file. Some operating systems do not support a function to flush file buffers. In those cases an implicit close and open takes place. When this occurs, all file and record locks will be released for the file.

for, break, continue, repeat

label	**`for`**	*numvar prep numexp1 prep numexp2*
label	**`for`**	*numvar prep numexp1 prep numexp2 prep numexp3*
label	**`break`**	
label	**`break`**	**`if`** *cond*
label	**`break`**	**`if not`** *cond*
label	**`break`**	**`if`** *function-key*
label	**`break`**	**`if not`** *function-key*
label	**`break`**	**`if`** *expression*
label	**`break`**	**`if not`** *expression*
label	**`continue`**	
label	**`continue`**	**`if`** *cond*
label	**`continue`**	**`if not`** *cond*
label	**`continue`**	**`if`** *function-key*
label	**`continue`**	**`if not`** *function-key*
label	**`continue`**	**`if`** *expression*
label	**`continue`**	**`if not`** *expression*
label	**`repeat`**	

numvar is the loop variable
numexp1 is the starting value operand
numexp2 is the ending value operand
numexp3 is the increment value operand
cond is one of **`equal`**, **`less`**, **`over`**, **`eos`** or **`greater`**
expression is an algebraic expression
function-key is one of **`F1`**, **`F2`**, **`F3`**, **`F4`**, **`F5`**, **`F6`**, **`F7`**, **`F8`**, **`F9`**, **`F10`**, **`F11`**, **`F12`**, **`F13`**, **`F14`**, **`F15`**, **`F16`**, **`F17`**, **`F18`**, **`F19`**, **`F20`**, **`up`**, **`down`**, **`left`**, **`right`**, **`insert`**, **`delete`**, **`home`**, **`end`**, **`pgup`**, **`pgdn`**, **`tab`**, **`bktab`**, **`esc`**, or **`enter`**

Flags affected: none

See also: loop

Statements between the for and repeat statements are executed zero or more times. The operands of the for statement control the number of times the statements are executed. The break and continue statements conditionally control execution of the program lines between the for and repeat statements.

One for statement is required before the break, continue, and repeat statements. One repeat statement is required after the for, break, and continue statements. The break and continue statements are optional and may be specified any number of times.

If the increment value operand is not specified, it defaults to one. The loop variable is set to the starting value before the first iteration through the loop is attempted.

If the increment value is positive and the loop variable is greater than the ending value, then the for/repeat loop terminates and execution continues at the statement after the repeat statement. If the increment value is negative and the loop variable is less than the ending value, then the for/repeat loop terminates and execution continues at the statement after the repeat statement. If neither of these cases is true, then statement execution continues at the statement after the for statement. When the corresponding repeat statement is executed, the increment value is added to the loop variable and the termination test is performed again. Execution continues as described above.

The break statement causes the for/repeat loop to terminate and execution to continue after the repeat statement. If the break statement is conditional, then this action occurs only if the condition is true.

The continue statement causes execution to continue immediately with the repeat statement. That is, the loop variable is increased incrementally, the termination test is performed, and program execution continues accordingly.

for/repeat loops may be nested up to 32 levels.

format

label `format` *exp prep charexp prep charvar*

exp is the source operand
charexp is the formatting mask
charvar is the destination operand

Flags affected: eos, over

The format statement causes numeric and character data to be reformatted. The formatting mask utilizes data from the source operand to determine the data stored in the destination.

Data is moved into the destination operand following the rules of the formatting mask. The actual data that is moved depends on the formatting mask. The contents of the source operand are not changed.

The formatting mask contains special control characters that denote how data is moved to the destination. The mask can contain any character, but only the control characters affect the movement of data. Formatting continues as follows: each character in the mask is compared with the control characters. If the character in the mask is not one of the control characters, then it is moved to the destination unchanged and no characters from the source are moved. Formatting continues with the next character in the mask.

The form pointer of the destination is set to one. The length pointer of the destination is set to the last character moved.

The eos flag is set if the destination is too small to hold all the characters moved. The over flag is set if the format mask string is erroneous.

The formatting of a character source uses different mask characters than those for the formatting of a numeric source.

Three control characters are applicable if the source operand is a character variable, literal, or expression:

A	One character is moved from the source.
~	One character from the source is ignored.
\	The following character in the mask is moved to the destination, even if it is a control character.

Fourteen control characters are applicable if the source operand is a numeric variable, literal, or expression:

Z	A digit or blank character is moved from the source to the destination.
9	A character is moved from the source. However, if the character is a blank, it is converted to a zero.
,	A comma is moved to the destination. However, if no digits have been moved, a blank is moved instead of a comma.
&	The character following this symbol is moved to the destination, unless no digits have been moved, in which case a blank is moved to the destination. Thus, **&**, is the same as , by itself.
(	This character is moved to the destination if the source value is negative; otherwise, a blank is moved.
)	This character is moved to the destination if the source value is negative; otherwise, a blank is moved.
-	A minus sign is moved to the destination if the source value is negative; otherwise, a blank is moved.
+	A plus sign or a minus sign is moved to the destination, depending on whether the source is positive or negative.

<	The character following this symbol replaces all blanks moved from the source, blanks created by the comma, or blanks created by the lack of a sign (including the blank created by the closing parenthesis). If the **<** control character is used in conjunction with a **9** control character, the **<** is ignored.
>	The character following this symbol is floated to the left of the first digit moved. If the **>** control character is used in conjunction with a **9** control character, the far left blank will be replaced with the character following the **>**.
$	The dollar sign character is floated to the left of the first digit or decimal point moved. This control character is equivalent to **>$**.
^	The caret symbol is used to define decimal point justification in the field. The following character is used as a replacement for the decimal point.
.	A period is used to define decimal point justification in the field. This control character is equivalent to **^.** .
~	One character from the source is ignored.
\	The following character in the mask is moved to the destination, even if it is a control character.

fposit

label **`fposit`** *file*`,` *numvar*
label **`fposit`** *ifile*`,` *numvar*
label **`fposit`** *afile*`,` *numvar*

file is the label of a file declaration
ifile is the label of an ifile declaration
afile is the label of an afile declaration
numvar is the destination operand

Flags affected: over

See also: reposit

The fposit statement stores the value of the current file position in the destination operand. If the value to be stored is too large for the destination variable, the over flag is set.

The fposit instruction is often used with the reposit instruction to restore the current file position to an earlier value.

get

label **`get`** *queue; list*

queue is the queue variable
list is the list of character and numeric variables

Flags affected: over

The get statement moves a message from the beginning of the specified queue into the list of variables. If there are no messages in the queue, the over flag is set and the variables in the list are cleared and zeroed.

getcolor

label **`getcolor`** *image*`;` *numvar*

image is the first operand
numvar is the destination operand

Flags affected: over

The color of the dot located at the current graphics draw position is moved to the destination operand. The number that represents the color is the 24 bit RGB value of the color.

In non-graphical versions of DB/C DX, execution of the getcolor statement results in an error.

If the value of the color is truncated while it is being moved, over is set.

getcursor

label `getcursor` *numvar1 prep numvar2*

numvar1 is the horizontal position operand
numvar2 is the vertical position operand

Flags affected: over

The position of the cursor with respect to the current subwindow is moved to the operands. The horizontal position of the cursor is moved to the horizontal position operand. The vertical position of the cursor is moved to the vertical position operand.

If either position is truncated while it is being moved, the over flag is set.

getendkey

label **`getendkey`** *numvar*

numvar is the destination operand

Flags affected: none

See also: keyin, setendkey, clearendkey

The getendkey statement moves the numeric value of the key that caused the most recently completed keyin statement to terminate to the destination operand. The ending key values are described in the Keyboard and Display Manipulation Statements section of the DB/C Programming Language General Information chapter.

If **enter** is specified as an ending key and if it is the key that terminated the keyin statement, then 256 is moved to the destination. The value zero is moved to the destination if a keyin timeout occurred or if keyin continuous was in effect and was the cause of the keyin statement termination.

getglobal

label **`getglobal`** *charexp prep adrvar*

charexp contains the name of the global variable
adrvar is the destination address variable

Flags affected: over, less

The getglobal statement moves the address of a global variable into the destination address variable. The getglobal statement is useful when the global variable was declared in a different module, or when the global variable was created in a different module using the makeglobal instruction.

The name of the global variable is specified by the first operand. If the variable does not exist, the address variable is set to invalid and the over flag is set.

The destination operand must be the **@** form of the same type of variable as the global variable or it must be a typeless address variable. If the type of the address variable is not the same type as the global variable, the less flag is set.

getlabel

label **`getlabel`** *charexp prep lblvar*

charexp contains the name of the external label
lblvar is the destination label variable

Flags affected: over

See also: external

The getlabel statement moves the address of an external label into the destination label variable.

The name of the external label is specified by the logical string of the first operand. If the label does not exist, the label variable is invalid and the over flag is set.

getmodules

label **`getmodules`** *chrarray*

chrarray is the destination character array variable

Flags affected: eos, less

See also: loadmod

The getmodules statement moves the names of all loaded modules and instances into the destination array. Each module/instance combination is moved to an element of the destination array. The first element of the array will contain the current module and the current instance. All instances for a particular module will be grouped together as contiguous elements of the array. The first instance for each module will be the current instance for that module. Each element is stored like this: *module***<***instance***>**.

The eos flag is set if a module/instance string is truncated. The less flag is set if there are not enough elements in the array to hold all the module/instance combinations.

getname

label `getname` *variable prep charvar*

variable is the source variable
charvar is the destination character variable

Flags affected: eos, over

See also: record, list

The getname statement moves the name of the source variable to the destination if that name is available. To make a name available, use the **with names** operand on a list or record statement.

If the name is not available, the over flag is set and the destination form and length pointers are set to zero. If the destination is not large enough to hold the name of the source variable, the name is truncated on the right and the eos flag is set.

getobject

label **`getobject`** *adrvar*

adrvar is the destination object address variable

Flags affected: none

See also: call, make

The getobject statement moves a pointer to the object variable that defines the instance of the class that has current runtime scope into the destination variable. If there is no current instance of a class, then the destination address variable is cleared.

getpaperbins

label **`getpaperbins`** *charaexp, chararray*

charaexp is the source operand
chararray is the destination

Flags affected: eos, less, over

The getpaperbins statement moves the names of the paper bins for a printer into the elements of the destination array. The source operand provides the name of the printer for which the paper bins are made available.

If the **dbcdx.print.destination=client** runtime property is set and Smart Client is being used, the list of paper bins is gotten for the printer on the client computer, not the server.

The over flag is set if the printer name is invalid or doesn't exist. The eos flag is set if any paper bin name is truncated. The less flag is set if there are not enough elements in the array to hold all the paper bin names.

getpapernames

label `getpapernames` *charaexp, chararray*

charaexp is the source operand
chararray is the destination

Flags affected: eos, less, over

The getpapernames statement moves the names of the paper available for a printer into the elements of the destination array. The source operand provides the name of the printer for which the paper names are made available.

If the **dbcdx.print.destination=client** runtime property is set and Smart Client is being used, the list of paper names is gotten for the printer on the client computer, not the server.

The over flag is set if the printer name is invalid or doesn't exist. The eos flag is set if any paper name is truncated. The less flag is set if there are not enough elements in the array to hold all the paper names.

getparm

label **`getparm`** *cadrvar*
label **`getparm`** *cadrvar*, *varvar1*
label **`getparm`** *cadrvar*, *varvar1*, *varvar2*
label **`getparm`** *cadrvar*, *varvar1*, *varvar2*, *varvar3*
label **`getparm`** *cadrvar*, *varvar1*, *varvar2*, *varvar3*, *varvar4*
label **`getparm`** *cadrvar*, *varvar1*, *varvar2*, *varvar3*, *varvar4*, *varvar5*

cadrvar is the keyword destination operand
varvar1 is the first typeless address variable destination
varvar2 is the second typeless address variable destination
varvar3 is the third typeless address variable destination
varvar4 is the fourth typeless address variable destination
varvar5 is the fifth typeless address variable destination

Flags affected: over

See also: verb, type, cverb

The getparm statement moves the address of the next parameter from the optional parameter list associated with a user-defined verb into the destination operand(s).

The address of the keyword is put into the keyword destination operand. All alphabetic characters in the character string that *cadrvar* points to will have been translated to upper case. If the parameter does not have a keyword, the address of a character variable with a zero form pointer is put into the keyword operand.

The address of the variable or literal value of the parameter (if it exists) is put into the first typeless address variable. The addresses of any additional values are put into the second, third, fourth, and fifth typeless address variables. Any unused typeless address variables are set to values that will cause the type operation to return zero.

If a getparm operation is attempted after the last parameter has been retrieved, the over flag is set and all operands are invalid.

getposition

label   **`getposition`** *image*`;` *numvar1*`,` *numvar2*

`getpos` may be used in place of **`getposition`**

image is the first operand
numvar1 is the horizontal position operand
numvar2 is the vertical position operand

Flags affected: over

The getposition statement moves the current graphics draw position to the operands. The horizontal position is moved to the horizontal position operand. The vertical position is moved to the vertical position operand.

In non-graphical versions of DB/C DX, execution of the getposition statement results in an error.

If either position is truncated while it is being moved, the over flag is set.

getprinters

label **`getprinters`** *chararray*

chararray is the destination

Flags affected: eos, less

The getprinters statement moves the names of the currently available printers into the destination array. Each printer element of the destination array contains one printer name.

If the **dbcdx.print.destination=client** runtime property is set and Smart Client is being used, the list of printers will contain only the printers available on the client not the server.

The eos flag is set if any printer name is truncated. less is set if there are not enough elements in the array to hold all printer names.

getwindow

label **`getwindow`** *numvar1 prep numvar2 prep numvar3 prep numvar4*

numvar1 is the top operand
numvar2 is the bottom operand
numvar3 is the left operand
numvar4 is the right operand

Flags affected: over

The getwindow statement moves the current subwindow size to the operands. The top line number is moved to *numvar1*. The bottom line number is moved to *numvar2*. The left column number is moved to *numvar3*. The right column number is moved to *numvar4*.

If any parameter is truncated while it is being moved, the over flag is set.

goto

label **`goto`** *prog-label*
label **`goto`** *prog-label* **`if`** *cond*
label **`goto`** *prog-label* **`if not`** *cond*
label **`goto`** *prog-label* **`if`** *function-key*
label **`goto`** *prog-label* **`if not`** *function-key*
label **`goto`** *prog-label* **`if`** *expressionfi*
label **`goto`** *prog-label* **`if not`** *expression*

prog-label is a program execution label or a program label variable
cond is one of **`equal`**, **`less`**, **`over`**, **`eos`** or **`greater`**
function-key is one of **`F1`**, **`F2`**, **`F3`**, **`F4`**, **`F5`**, **`F6`**, **`F7`**, **`F8`**, **`F9`**, **`F10`**, **`F11`**, **`F12`**, **`F13`**, **`F14`**, **`F15`**, **`F16`**, **`F17`**, **`F18`**,**`F19`**, **`F20`**, **`up`**, **`down`**, **`left`**, **`right`**, **`insert`**, **`delete`**, **`home`**, **`end`**, **`pgup`**, **`pgdn`**, **`tab`**, **`bktab`**, **`esc`**, or **`enter`**
expression is an algebraic expression

Flags affected: none

See also: setendkey

The goto statement causes program execution to continue conditionally or unconditionally at the statement specified by *prog-label*. If *prog-label* is a program execution label, execution will continue at the statement with that label. If *prog-label* is a program label variable, execution will continue at the statement with the label whose value has been most recently assigned to the label variable.

If the first format is used, program execution unconditionally continues at the statement specified by *prog-label*. If one of the other formats is used, program execution continues at the statement specified by *prog-label* only if the condition tested by an if is true or the condition tested by an if not is false. If a condition is not met, program execution continues with the statement that follows the goto statement.

The function-key form of the goto statement is only applicable following a keyin statement that was interrupted when a function key was pressed. Each *function-key* condition may only be tested once with a goto, call, or return operation. All *function-key* conditions are reset after they are checked and at the start of keyin execution.

hide

label **`hide`** *resource*
label **`hide`** *image*

resource is the source operand
image is the source operand

Flags affected: none

See also: show

The resource or image specified as the source operand is removed from view.

In non-graphical versions of DB/C DX, execution of the hide statement with an image variable results in an error.

if, else, endif

label `if` *cond*
label `if not` *cond*
label `if` *function-key*
label `if not` *function-key*
label `if` *expression*
label `if not` *expression*
label `else`
label `else` `if` *cond*
label `else` `if not` *cond*
label `else` `if` *function-key*
label `else` `if not` *function-key*
label `else` `if` *expression*
label `else` `if not` *expression*
label `endif`

cond is one of equal, less, over, eos or greater
function-key is one of `F1`, `F2`, `F3`, `F4`, `F5`, `F6`, `F7`, `F8`, `F9`, `F10`, `F11`, `F12`, `F13`, `F14`, `F15`, `F16`, `F17`, `F18`,`F19`, `F20`, `up`, `down`, `left`, `right`, `insert`, `delete`, `home`, `end`, `pgup`, `pgdn`, `tab`, `bktab`, `esc`, or `enter`
expression is an algebraic expression

Flags affected: none

See also: setendkey

The if, else, and endif statements conditionally control execution of subsequent lines of code. These operations and the statements enclosed by these operations are called an if-endif construct.

The if statement is used to test whether a condition is true. If the condition is true (that is, if the function key has been pressed, if the algebraic expression evaluates to a value other than zero, or if the flag is set), then execution continues with the next statement. Otherwise, execution continues after the else statement if it exists, or after the endif statement if the else statement does not exist.

The if not statement is used to test whether a condition is false. If the condition is false, then execution continues with the next statement. Otherwise, execution continues after the else statement if it exists, or after the endif statement if the else statement does not exist.

The else statement is optional. The else statement causes the program to execute the next statement if the condition being tested by the previous if statement or if not state ment is not satisfied.

An endif statement is required whenever an if statement is present. The endif statement denotes the end of the if-endif construct. After the statements in the if construct have been executed, the program continues execution with the line following the endif statement.

The else if statement is analogous to an else statement immediately followed by an if statement. The else if not statement is analogous to an else statement immediately followed by an if not statement. The else if notations are useful when testing for multiple conditions within a single if-endif construct. The else if notations do not require any additional endif statements.

insert

label **`insert`** *ifile, charvar*
label **`insert`** *afile*

ifile is the label of an ifile declaration
afile is the label of an afile declaration
charvar is the key

Flags affected: none

The insert statement adds the specified key to an index file or associative index file without altering the data file.

The insert operation adds a key to an indexed file without changing the logical records of the data file. The logical string of *charvar* is used as the key.

For an index file, the logical string from the character variable is used as the new key to be put into the index. If duplicates are not allowed (that is, dup is not in the ifile declaration), then the index is searched for a key that matches the new key. If a match is found, then the duplicate key error occurs. If no match is found or if duplicates are allowed, then the new key is inserted into the index. The record that is associated with that key is the last record read or written through any ifile or afile variable.

The insert operation places the key information of the most recently read or written record into the aimdex of the afile.

For an aimdex file, the field information specified in the aimdex utility parameter list is used to obtain the key information. The insert statement places this key information into the aimdex file. The record that is associated with that key is the last record read or written through any file, ifile, or afile variable.

keyin

label `keyin` *list*

list is a comma delimited list that contains variables, literals, octal and hexadecimal constants, display control codes, and keyin control codes

Flags affected: varies

See also: display, clearendkey, getendkey, setendkey

The keyin statement accepts input from a keyboard. The keyin statement typically displays the characters on the screen and alters variables in the program.

When a variable is encountered in the list, the cursor appears at the current cursor position and the program waits for characters to be typed on the keyboard. Each valid character entered is displayed at the cursor position. The cursor position is increased incrementally after each character is entered.

Control codes modify the manner in which data is requested and entered into variables.

When a character variable is encountered in the operand list, each character entered from the keyboard is placed into the variable starting at the first character position and continuing through the maximum length of the variable. Each character entered is displayed on the screen. The cursor position is increased incrementally for each character entered. Entry of characters is terminated by pressing the enter key or an end key. If more characters are entered than will fit in the character variable, only those characters that will fit are accepted and a beep is sounded. The form pointer of the variable is set to one and the logical length pointer is set to point to the last character entered before the enter or the end key was pressed. Blanks are stored after the last character entered up through the physical length of the variable. If no characters are entered and just the enter key or an end key is pressed, the form pointer of the variable is set to zero.

When a numeric variable is encountered in the operand list, only characters that comprise a valid DB/C number may be entered. The number of characters accepted is the total length of the numeric variable including the decimal point. Only the correct number of digits to the left or right of the decimal point is allowed to be entered. When the enter key or an end key is pressed, the characters entered are moved to the numeric variable. The format of the numeric variable is preserved. If any extra characters are entered, a beep is sounded and the character is ignored. If no characters are entered and just the enter key or an end key is pressed, the numeric variable is set to zero.

After an active end key other than the enter key is pressed, keyin of the current variable and processing of the keyin statement are terminated. See setendkey and clearendkey for a description of how to alter the active end keys. Any variables remaining in the operand list are processed as though the enter key was pressed for each.

When a literal is encountered in the operand list, the literal value is displayed on the screen starting at the current cursor position. The cursor position is increased incrementally for each character entered.

When octal or hexadecimal constants are encountered in the operand list, the character represented by that character code is displayed in the same manner as a single character in a character variable.

If the cancel key is pressed during entry of a keyin variable, all characters typed into the field up to that point are erased and the cursor is placed at the beginning of the field. If no characters have been typed into the current field when the cancel key is pressed, only a beep sounds.

If the backspace key is pressed during entry of a keyin variable, the previous character typed into the field is erased and the cursor is moved left one position. If no characters have been typed into the current field when the backspace key is pressed, the backspace is ignored.

All display control codes work the same way with a keyin statement except for **`*+`**, **`*-`**, and **`*zf`**. These three exceptions are discussed in the following sections.

`*edit` is the edit control code. This control code affects the next variable encountered in the operand list. If the next variable is a numeric variable, then the variable is displayed and the cursor is positioned at the first character displayed. If any character other than an ending key is pressed, the field is first cleared; otherwise, the value in the variable is retained. If the next variable is a character variable, then special editing takes place. First, the value of the variable is displayed. Blanks are displayed for the number of characters that are between the length pointer of the variable and the maximum length of the variable. The cursor is positioned at the first character displayed. Characters typed at the keyboard cause the typed character to be inserted into the field and the cursor moved to the right one position. If ***ovsmode** is in effect, the typed character replaces the character at the cursor position. The following keys have special functions: left, right, back space, and delete. Left and right are non-destructive cursor movement keys. The cursor is only allowed to move within the field itself. Back space works normally, except all characters to the right of the character removed move left one character position. The enter key or an end key completes the keyin. When keyin is finished, the form pointer is set to one and the logical length is set to the number of characters in the variable. The edit insert/overstrike is controlled by ***insmode** and **ovsmode*. The **dbcdx.keyin.editmode** runtime property modifies this action.

***editon** is the edit on control code. This control code works exactly like ***edit** except it remains in effect until it is cancelled by the ***editoff** control code.

***editoff** is the control code that cancels the effect of the ***editon** control code.

***insmode** causes editing of variables to take place in insert mode. In insert mode, a character typed at the keyboard is inserted at the current cursor position. All the characters in the field that lie to the right of the current cursor position are moved to the right one position. This is the default editing mode. This control code is only applicable if ***edit** or ***editon** is in effect.

***ovsmode** causes editing of variables to take place in over strike mode. In over strike mode, every character typed at the keyboard overwrites the character that is at the current cursor position. The characters to the right of the current cursor position are unchanged. This control code is only applicable if ***edit** or ***editon** is in effect.

kl=charvar***:***n* is the keyin limited field control code. This control code affects the character variable designated by *charvar*. When the ***kl** control code is in effect, keyin of *charvar* is conducted using the same keystrokes as in ***edit** with character variables. However, the field length on the screen is limited to *n* characters, where *n* is a decimal constant or a numeric variable. Only *n* characters keyed in will be displayed on the screen.

***de** is the digit entry control code. This control code affects only the next variable encountered in the operand list. This control code only allows entry of the digits **0** through **9**. Typing any other character causes a beep to sound and the character to be ignored.

***zf** is the zero fill control code. This control code affects only the next variable encountered in the operand list. This control code only applies to character variables. This control code causes all blanks in the variable to be replaced with zeros.

***j**r is the justify right control code. This control code affects only the next variable encountered in the operand list. This control code only applies to character variables. This control code causes the characters entered to be right justified in the variable.

***jl** is the justify left control code. This control code affects only the next variable encountered in the operand list. This control code causes the characters entered to be left justified in the variable.

***clickon** is the control code that causes a click to sound every time a character is entered. This option is unavailable on some terminals. This code is canceled by the ***clickoff** code or by chaining to another program.

***clickoff** is the control code that cancels the action of the ***clickon** control code.

***eoff** is the echo off control code. This control code affects all variables that follow in the operand list. Nothing is displayed and the cursor is not moved as each character is typed on the keyboard.

***eon** is the echo on control code. This control code reverses the action of the ***eoff** control code and causes the system to revert to normal display of characters as they are typed.

***eson** is the echo secret on control code. This code affects all variables that follow in the operand list. As each character is typed in the keyboard, an asterisk or a character specified in the ***eschar** control code is displayed on the screen instead of the character that was typed.

***esoff** is the echo secret off control code. This code cancels the effect of the ***eson** control code.

eschar=charexp* is the echo secret character control code. The form pointed character in *charexp* is the character that replaces the asterisk display as described in the ***eson** control code.

***uc** is the upper-case control code. This control code causes the characters typed at the terminal to be converted to upper-case whether the shift key is pressed or not. This control code is canceled by the ***lc**, ***it**, or ***in** control codes or by chaining to another program.

***lc** is the lower-case control code. This control code causes the characters typed at the terminal to be converted to lower-case whether the shift key is pressed or not. This control code is canceled by the ***uc**, ***it**, or ***in** control codes or by chaining to another program.

***it** is the invert keyboard to lower-case control code. When this code is in effect, the keyboard is in upper-case mode. If a character is typed with the shift key pressed, the character is lower-case. Otherwise, the character is in upper-case. This control code is canceled by the ***uc**, ***lc**, or ***in** control codes, or by chaining to another program.

***in** is the invert to normal control code. This control code cancels the effects of the ***it**, ***uc**, and ***lc** control codes.

***cl** is the clear key ahead buffer control code. This control code discards any keystrokes that have been typed ahead.

***dv** is the display variable control code. This control code affects only the next variable or literal encountered in the operand list. This control code causes that variable or literal to be displayed as it would be by the display operation instead of being used as an input variable.

***dcon** in a keyin operation is the enter comma control code. This control code causes a comma to be accepted instead of a decimal point (period) in numeric variables. This control code affects all numeric variables that follow. It is cancelled by the ***dcoff** control code or by chaining to another program. It also affects the operation of the display statement.

***dcoff** is the cancel comma control code. This control code cancels the effect of the ***dcon** control code.

***kcon** or ***+** in a keyin statement is the keyin continuous control code. This control code affects all variables that follow in the operand list. The keyin of each variable does not have to be terminated with the enter key. After the last character is typed into a variable, the system continues the keyin of the next variable as if the enter key had been pressed.

***kcoff** or ***–** in a keyin statement reverses the action of ***kcon**. This control code causes the system to revert to the normal mode of requiring that the enter key be pressed in order to enter typed data into a variable.

t**, ***t=n* and ***t***n* are the timeout control codes. These control codes affect keyin of all variables that follow in the operand list. The *n* in the ***t=***n* format is a numeric variable or a decimal constant with a value between 0 and 30000 and which is the timeout interval in seconds. The *n* in the ***t***n* format is a decimal constant with a value between 1 and 254, inclusive, and which is the timeout interval in seconds. When the ***t** format is specified the timeout interval is two seconds. The timeout action occurs when the time between two keystrokes during entry of a variable is greater than the specified timeout interval. When the timeout interval is zero, a timeout occurs unless the characters have already been typed (that is, typed ahead). The effect of a timeout is analogous to pressing the enter key for the current variable and all remaining variables in the operand list. The current variable contains any characters already entered and remaining variables are zeroed or cleared.

***toff** or ***t255** is the cancel timeout control code. These control codes cancel the timeout feature and revert to normal data entry with no timeout in effect.

***rv** is the retain variable control code. This control code affects only the next variable encountered in the operand list. If there are no characters entered into the variable when the enter key is pressed, then the entry is considered to be a null entry. If there were characters entered, then characters are stored into the variable normally. If there is a null entry, the previous value of the variable is retained instead of being cleared or set to zero. The eos flag is set with a null entry. If the entry is not null, the eos flag is cleared. When the ***t** control code is also in effect for this variable, the less flag is set if a timeout occurred and the less flag is cleared if a timeout did not occur. If the keyin statement was terminated by the pressing of a function key, the over flag is set; otherwise, the over flag is cleared.

lcmove

label **`lcmove`** *charvar1 prep charvar2*

charvar1 is the source operand
charvar2 is the destination operand

Flags affected: eos

The lcmove statement moves the length-pointed character in the source operand to the first character position in the destination operand. No changes are made to the form pointers or logical length pointers of either operand. If the logical length pointer of the source is zero, then no move takes place and the eos flag is set.

lenset

label **`lenset`** *charvar*

charvar is the destination operand

Flags affected: none

See also: endset

The lenset statement sets the length pointer of the destination to the same value as its form pointer.

link

label **`link`** *resource prep queue*
label **`link`** *device prep queue*

resource is the source operand
device is the source operand
queue is the queue variable

Flags affected: none

The resource or device specified by the source operand is linked to the queue variable. The resource or device will either provide queue messages or receive queue messages.

load

label	`load`	*var prep numexp prep list*
label	`load`	*charvar prep device*
label	`load`	*image prep device*

var is the destination variable
numexp is the index
list is a list of literals, character variables, numeric variables, arrays or list variables
charvar is the destination operand
image is the destination operand
device is the source operand

Flags affected: equal, less, over (both operands numeric); eos (one or both operands character); none (image and device variables are operands)

See also: move, store

In the first form, the load statement moves one of the variables in the list to the destination. The value of the index (*numexp*) determines which member of the list is moved. Each element of an array or list variable is counted as one variable.

The source is the Nth item from the list where the index defines N. The index is truncated to an integer value, not rounded. If the value N is less than one or greater than the number of variables in the list, then the load statement does nothing.

After an item is selected as the source, an implicit move operation is performed as the item is moved to the destination. Flags are set according to the rules used with the move instruction.

In the second form, the load statement moves text from the source device to the character variable.

In the third form, the load statement moves an image from the source device to the image variable. In non-graphical versions of DB/C DX, this form of load statement causes a runtime error.

loadadr

label **`loadadr`** *adrvar prep numexp prep list*
label **`loadadr`** *adrvar prep numexp prep lstvar*
label **`loadadr`** *adrvar prep numexp prep array*

adrvar is the destination operand
numexp is the index
list is a list of source variables
lstvar is a list variable
array is an array variable

Flags affected: none

The source variables may be any type of variable except a label variable. The index (*numexp*), used to specify the source variables, is truncated to an integer value, not rounded. If the index is less than one or greater than the number of variables in the list, then the loadadr operation does nothing.

If the first format is used, the loadadr statement moves the address of a specified variable from the list of variables to the destination. The destination operand (*adrvar*) must be the **@** form of the selected variable or a typeless address variable. The value of the index (*numexp*) determines which variable in the list is the destination. A list variable or an array counts as one variable and must respectively have a list **@** or an array **@** destination variable (*adrvar*).

If the second or third format is used, the loadadr statement moves the address of a specified variable within the given list or array variable (*lstvar* or *array*) to the destination. The value of the index (*numexp*) determines which variable within the given list or which element within the given array is specified. The destination operand must be the **@** form of the selected variable within the list/array or a var **@** variable.

loadlabel

label `loadlabel` *lblvar prep numexp prep list*

lblvar is the destination operand
numexp is the index
list is a list of program labels and label variables

Flags affected: none

See also: movelabel, movelv, movevl, storelabel

The loadlabel statement moves one of the entries from the list to the destination label variable. The value of the index determines which member of the list is moved to the destination variable. The source is the N^{th} program label or variable from the list where the index defines N. The index is truncated to an integer value, not rounded. If N is less than one or greater than the number of items in the list, then loadlabel does nothing.

loadmod

label **`loadmod`** *charexp*

charexp is the source operand

Flags affected: none

See also: chain

The loadmod statement loads a file as a secondary program module. If the source operand is a character variable, the logical string is used as the file name. If the file is not found or is otherwise invalid, an error occurs. The secondary module remains loaded until a chain operation executes.

If the logical string of the source operand terminates with **<***name***>**, then name specifies the secondary module instance that will be made current. **<***name***>** is not part of the file name that is loaded. If the same module, but a different instance, has previously been loaded, then a new copy of the data area is created. If the same module, same instance, has previously been loaded, then the second instance is made current.

Module instance names must be eight characters or less.

If the logical string contains **`< >`** or does not contain **<***name***>**, then the module being loaded or made current is called the unnamed instance of a secondary module. Both named and the unnamed instances of a secondary module may exist at the same time.

loadparm

label	**`loadparm`**	*numvar prep cadrvar*
label	**`loadparm`**	*numvar prep cadrvar prep varvar1*
label	**`loadparm`**	*numvar prep cadrvar prep varvar1, varvar2*
label	**`loadparm`**	*numvar prep cadrvar prep varvar1, varvar2, varvar3*
label	**`loadparm`**	*numvar prep cadrvar prep varvar1, varvar2, varvar3, varvar4*
label	**`loadparm`**	*numvar prep cadrvar prep varvar1, varvar2, varvar3, varvar4, varvar5*

numvar is the index

cadrvar is the keyword destination operand

varvar1 is the first typeless address variable destination

varvar2 is the second typeless address variable destination

varvar3 is the third typeless address variable destination

varvar4 is the fourth typeless address variable destination

varvar5 is the fifth typeless address variable destination

Flags affected: over

See also: getparm, resetparm, verb, cverb

The loadparm statement performs the getparm operation from the Nth optional parameter, where the index (*numvar*) defines N. The index is truncated to an integer value, not rounded. If the value N is less than one or greater than the number of optional parameters, the over flag is set and all destination operands are invalid.

loop, break, while, until, continue, repeat

label **`loop`**
label **`break`**
label **`break if`** *cond*
label **`break if not`** *cond*
label **`break if`** *function-key*
label **`break if not`** *function-key*
label **`break if`** *expression*
label **`break if not`** *expression*
label **`while`** *cond*
label **`while not`** *cond*
label **`while`** *function-key*
label **`while not`** *function-key*
label **`while`** *expression*
label **`while not`** *expression*
label **`continue`**
label **`continue if`** *cond*
label **`continue if not`** *cond*
label **`continue if`** *function-key*
label **`continue if not`** *function-key*
label **`continue if`** *expression*
label **`continue if not`** *expression*
label **`until`** *cond*
label **`until not`** *cond*
label **`until`** *function-key*
label **`until not`** *function-key*
label **`until`** *expression*
label **`until not`** *expression*
label **`repeat`**

The until and while statements may be combined with the repeat statement on one program line. The repeat statement is positioned first and the until or while statement follows.

The until and while statements may be combined with the loop statement on one program line. The loop statement is positioned first and the until or while statement follows.

cond is one of equal, less, over, eos, or greater

function-key is one of **`F1`**, **`F2`**, **`F3`**, **`F4`**, **`F5`**, **`F6`**, **`F7`**, **`F8`**, **`F9`**, **`F10`**, **`F11`**, **`F12`**, **`F13`**, **`F14`**, **`F15`**, **`F16`**, **`F17`**, **`F18`**,**`F19`**, **`F20`**, **`up`**, **`down`**, **`left`**, **`right`**, **`insert`**, **`delete`**, **`home`**, **`end`**, **`pgup`**, **`pgdn`**, **`tab`**, **`bktab`**, **`esc`**, or **`enter`**

expression is an algebraic expression

Flags affected: none

See also: for

The break, continue, while, and until statements conditionally control execution of the program lines between loop and repeat statements.

The loop statement defines the beginning of the construct. Each loop statement must have a corresponding repeat statement.

When a loop statement is encountered, execution of the following program lines continues until a repeat statement is encountered. Then execution continues at the statement following the loop statement.

The break and continue instructions are optional. Any number of break and continue statements can be placed between the loop and repeat statements. If the expression following break is true, then execution continues at the statement following the repeat statement. If the expression following continue is true, then execution continues at the statement following the loop statement.

The until and while instructions are optional. Any number of until and while statements can be placed between the loop and repeat statements. The until statement functions in the same way as the break statement. The while statement functions in the same way as until, except the condition testing is reversed (that is, if the expression following the while is false, then execution continues at the statement following the repeat statement).

The until and while statements may be combined with the loop statement on one line. The until and while statements may also be combined with the repeat statement on one line. The until and while statements will function in the same manner as when they are freestanding statements.

loop/repeat constructs may be nested up to 32 levels.

make

label **`make`** *objvar prep class*
label **`make`** *objvar prep class prep list*

objvar is the object variable
class is the class name
list is a list of variables, literals, and expressions

Flags affected: none

See also: call, destroy

The make statement instantiates the object variable as an object of the class specified by the class operand. This process causes a new set of data variables of the class and all ancestor classes to come into existence. These variables are used by methods that are called using the object variable.

If there is a make routine specified for the class or for any ancestor of the class, then each of those make routines are called successively. The deepest ancestor make routine is called first and the rest of the make routines are called in order of inheritance with the make routine of class called last. If the prep list form of the make routine is specified, then the variables, literals and expressions of the list are passed as parameters to each of the make routines. If any of the make routines can not be found, then an E 562 error occurs.

makeglobal

label **`makeglobal`** *charexp prep adrvar*

charexp is the name of a global variable
adrvar is the destination address variable

Flags affected: less

See also: getglobal

The makeglobal statement dynamically creates a global variable and moves its address to the destination address variable.

The first operand specifies the name and type of the global variable. The logical string of the first operand is of the form:

name`:`*type size*

name is the 1 to 31-character variable name.
type is **C**, **N**, **D**, **R**, **I**,or **O**.

C means create a character variable. **N** means create a numeric variable. **D** means create a device variable. **R** means create a resource variable. **I** means create an image variable. **O** means create an object variable.

size is specified only if type is **C**, **N**, or **I**. For type **C**, *size* must be an integer value from 1 to 65500. For type **N**, *size* must be either *n* or *n*`.`*m* where *n* is the size to the left of the decimal point and *m* is the size to the right of the decimal point. For type **I**, the size must be either *h*`,`*v* or *h*`,`*v*`,`*c* where *h* is the horizontal size, *v* is the vertical size and *c*, if specified, is the number of color bits per pixel.

For type **C** and type **N**, *size* may also contain an array specification which follows the variable size definition. The array specification is of the form `[`*n*`]`, `[`*n*`,`*n*`]`, or `[`*n*`,`*n*`,`*n*`]`.

The destination operand must be the **@** form of the same type of variable as the global variable or it must be a typeless address variable. The less flag is set if the type is invalid, the size is invalid, or if the type of the address variable is not the same as the global variable.

makevar

label ***makevar*** *charexp prep adrvar*

charexp defines a local unnamed variable
adrvar is the destination address variable

Flags affected: less

The makevar statement dynamically creates a local unnamed variable and moves its address to the destination address variable. The variable is contained in the current instance of the current module. It will be destroyed when the current instance is destroyed with an unload statement or by the chain statement.

The logical string of the first operand is of the form:

type size

type is **C**, **N**, **D**, **R**, **I**,or **O**.

C means create a character variable. **N** means create a numeric variable. **D** means create a device variable. **R** means create a resource variable. **I** means create an image variable. **O** means create an object variable.

size is specified only if type is **C**, **N**, or **I**. For type **C**, *size* must be an integer value from 1 to 65500. For type **N**, *size* must be either *n* or *n* **.** *m* where *n* is the size to the left of the decimal point and *m* is the size to the right of the decimal point. For type **I**, the size must be either *h* **,** *v* or *h* **,** *v* **,** *c* where *h* is the horizontal size, *v* is the vertical size and *c*, if specified, is the number of color bits per pixel.

For type **C** and type **N**, *size* may also contain an array specification which follows the variable size definition. The array specification is of the form **[***n***]**, **[***n***,***n***]**, or **[***n***,***n***,***n***]**.

The destination operand must be the **@** form of the same type of variable as the local unnamed variable or it must be a typeless address variable. The less flag is set if the type is invalid, the size is invalid, or if the type of the address variable is not the same as the local unnamed variable.

match

label **`match`** *charexp prep charvar*

charexp is the source operand
charvar is the destination operand

Flags affected: equal, less, eos

The match statement compares two character strings. The logical string of the source is compared with the logical string of the destination on a character-by-character basis. The comparison continues until one of the logical strings is exhausted or until the two characters being compared do not match. The operands remain unchanged.

If all characters match, equal is set and the lengths of the source and destination strings are compared. If the source string is longer, less is set.

If any of the characters do not match, equal is cleared and the values of the mismatched characters are compared. If the character from the source has a higher value, less is set.

The less flag has two different meanings, depending on the setting of equal. If equal is set, then less represents the comparative lengths of the two operands. If equal is cleared, then less represents the comparative character values of the two operands.

If the form pointer of either source or destination is zero, eos is set and no comparison is made.

Miscellaneous Arithmetic Statements

label	**`abs`**	*numexp1 prep numvar1*
label	**`abs`**	*numexp1* **`giving`** *numvar1*
label	**`arccos`**	*numexp1 prep numvar1*
label	**`arccos`**	*numexp1* **`giving`** *numvar1*
label	**`arcsin`**	*numexp1 prep numvar1*
label	**`arcsin`**	*numexp1* **`giving`** *numvar1*
label	**`arctan`**	*numexp1 prep numvar1*
label	**`arctan`**	*numexp1* **`giving`** *numvar1*
label	**`cos`**	*numexp1 prep numvar1*
label	**`cos`**	*numexp1* **`giving`** *numvar1*
label	**`exp`**	*numexp1 prep numvar1*
label	**`exp`**	*numexp1* **`giving`** *numvar1*
label	**`log`**	*numexp1 prep numvar1*
label	**`log`**	*numexp1* **`giving`** *numvar1*
label	**`log10`**	*numexp1 prep numvar1*
label	**`log10`**	*numexp1* **`giving`** *numvar1*
label	**`power`**	*numexp1 prep numexp2* **`,`** *numvar1*
label	**`power`**	*numexp1 prep numexp2* **`giving`** *numvar1*
label	**`sin`**	*numexp1 prep numvar1*
label	**`sin`**	*numexp1* **`giving`** *numvar1*
label	**`sqrt`**	*numexp1 prep numvar1*
label	**`sqrt`**	*numexp1* **`giving`** *numvar1*
label	**`tan`**	*numexp1 prep numvar1*
label	**`tan`**	*numexp1* **`giving`** *numvar1*

numexp1 is the source operand
numexp2 is the second source operand
numvar1 is the destination operand

Flags affected: equal, less, over

For the trigonometric functions, the specified operation is performed on the source operand and the result is placed in the destination operand. The value of the source operand is in radians.

For the exponential functions, the specified operation is performed on the source operand and the result is placed into the destination operand.

For the power statement, the first source operand is raised to the power of the second source operand, and the result is placed into the destination operand.

mod

label **`mod`** *numexp1 prep numvar1*
label **`mod`** *numexp1 prep numexp2 giving numvar1*

numexp1 is the first source operand
numexp2 is the second source operand
numvar1 is the destination operand

Flags affected: equal, less, over

If the first format is used, then the absolute value of the destination operand is divided by the absolute value of the source operand. The integer remainder is placed in the destination operand.

If the second format is used, then the absolute value of the second source operand is divided by the absolute value of the first source operand. The integer remainder is placed in the destination operand.

If either the first or second operand is fractional, the value is obtained by truncation, not rounding.

move

label **`move`** *exp prep list*
label **`move`** *exp prep lstvar*
label **`move`** *exp prep array*
label **`move`** *array1 prep array2*
label **`move`** *lstvar1 prep lstvar2*

exp is the source operand
list is the destination list of variables and arrays
lstvar is the destination list variable
array is the destination array variable
array1 is the source character or numeric array variable
array2 is the destination character or numeric array variable
lstvar1 is the source list variable
lstvar2 is the destination list variable

Flags affected: equal, less, over, eos (both operands numeric); eos (one or both operands character or both operands list variables)

The move statement transfers the contents of the source operand to each destination variable.

The destination list may consist of character variables or numeric variables. Numeric and character variables may not be mixed in the destination list.

If the first format is used, then the source operand is moved to each variable in the destination list. If an expression is the source, then the result of the evaluation of the expression is moved to each variable in the list.

If the second or third format is used, then the source operand is moved to each variable in the destination list or the destination array.

If the fourth format is used, the arrays must be exactly the same size (both in number of dimensions and in number of elements in each dimension). Each element of the source array is moved to the corresponding element in each destination array.

If the fifth format is used, then each individual variable from the source list is moved to the corresponding variable in the destination list. Any variable other than a numeric or character variable is ignored. Nested lists are expanded. Arrays are treated as a single variable. If any character destination variable value is truncated, eos is set.

If a destination variable is the same as the source variable, then the move occurs as if the variables were not the same variables.

For character destination variables, the characters in the logical string of the source are moved one at a time. Numeric data that is to be moved to character destination variables is treated as logical strings of characters. The characters are placed in the destination character variable starting in the first physical position. The destination form pointer is set to one and the logical length pointer is set to the number of characters moved. The eos flag is set if the logical string of the source will not fit into the destination variable. In this case, as many characters that will fit into the destination variable are moved.

If the source is a character variable whose form pointer is zero and the destination is a character variable, then the destination form pointer is set to zero and eos is cleared.

If source and destination are numeric, the source is rounded to fit each destination variable before being moved. equal, less, and over are set just as if an arithmetic operation took place.

If the source is a character variable and the destination is numeric, the format of the source must be a valid DB/C number. If the source does not conform to the rules defining valid numeric data, nothing is moved and eos is set. If the source is a valid number, it is rounded to fit each destination variable before being moved. If any significant digits or the minus sign is lost by truncation, the eos flag is set. Otherwise the eos flag is cleared. If the source logical string is null, the eos flag is cleared and no change is made to the destination. The equal, less, and over flags are not affected.

moveadr

label **`moveadr`** *var prep adrvar*
label **`moveadr`** *charlit prep adrvar*

var is the source operand
adrvar is the destination operand
charlit is the source operand

Flags affected: none

See also: loadadr, storeadr

The address of the source variable is moved into the destination.

In the first form, the source operand may be any type of variable except a label variable. The destination operand must be the **@** form of the same type of variable as the source or it must be a typeless address variable.

In the second form, the source operand is a character literal. The destination operand must be a character address variable or it must be a typeless address variable.

movefptr

label **`movefptr`** *charvar prep numvar*

charvar is the source operand
numvar is the destination operand

Flags affected: equal, over

The movefptr statement determines the value of the form pointer. The value of the form pointer in the source operand is moved to the destination operand. If the value of the pointer is zero, equal is set. If the value of the form pointer is truncated while being moved to the destination variable, over is set.

movelabel

label **`movelabel`** *lblvar1 prep lblvar2*
label **`movelabel`** *prog-label prep lblvar2*

lblvar1 is the source label variable
prog-label is the source program label
lblvar2 is the destination label variable

Flags affected: none

See also: loadlabel, movelv, movevl, storelabel

The address of the program execution label referred to in the source operand is moved to the destination label variable.

movelength

label **`movelength`** *var prep numvar*

var is the source operand
numvar is the destination

Flags affected: equal, over

The maximum length (possibly changed by sformat or nformat) of the source operand is moved to the destination. The source operand may be a character or numeric variable. If the value moved is zero, equal is set. If the maximum length value is truncated while being moved to the destination numeric variable, over is set.

movelptr

label **`movelptr`** *charvar prep numvar*

charvar is the source operand
numvar is the destination operand

Flags affected: equal, over

The movelptr statement determines the value of the length pointer. The value of the length pointer in the source operand is moved to the destination operand. If the value of the pointer is zero, equal is set. If the value of the length pointer is truncated while being moved to the destination variable, over is set.

movelv

label **`movelv`** *lblvar prep varvar*
label **`movelv`** *prog-label prep varvar*

lblvar is the source label variable
prog-label is the source program label
varvar is the destination variable

Flags affected: none

See also: movelabel, movevl

The address of the program execution label referred to by the source operand is moved to the destination variable. The destination variable must be a typeless (var) address variable.

movesize

label **`movesize`** *var prep numvar*

var is the source operand
numvar is the destination variable

Flags affected: equal, over

If the source operand is a character variable, the logical length of the source character variable is moved to the destination. If the value moved is zero, the equal flag is set. If the logical length value is truncated while being moved to the destination numeric variable, the over flag is set. If the source operand is a numeric variable, the number of digits to the left side of the decimal point is *left-side* and the number of digits after the decimal point is *right-side*. If there is no decimal point, *right-side* is zero. If the number of digits to the right of the decimal point is greater than nine, then *right-side* is set to 9. The value *right-side*`.`*left-side* is moved to the destination. If the value is truncated when it is moved, the over flag is set. The equal flag is always cleared.

movevl

label **`movevl`** *varvar prep lblvar*

varvar is the source operand
lblvar is the destination variable

Flags affected: none

See also: movelabel, movelv

The source variable must be a typeless (var) address variable that contains the address of a program execution label. This address is moved to the destination label variable.

multiply

label **`multiply`** *numexp1 prep numvar*
label **`multiply`** *numexp1 prep numexp2* **`giving`** *numvar*
label **`multiply`** *numexp1 prep numarray3*
label **`multiply`** *numarray1 prep numvar*
label **`multiply`** *numarray1 prep numarray3*
label **`multiply`** *numarray1 prep numarray2* **`giving`** *numarray3*

`mult` may be used in place of **`multiply`**

numexp1 is the first source operand
numexp2 is the second source operand
numvar is the destination operand
numarray1 is the first source operand
numarray2 is the second source operand
numarray3 is the destination operand

Flags affected: equal, less, over

If the first format is used, then the source operand is multiplied by the destination operand, and the result is placed in the destination operand.

If the second format is used, then the first source operand is multiplied by the second source operand, and the result is placed in the destination operand.

If the third format is used, the source operand is multiplied by each element of the destination operand.

If the fourth format is used, each element of the source array is multiplied together by the destination operand. The result is placed in the destination operand.

If the fifth format is used, each element of the source array is multiplied by the corresponding element in the destination array.

If the sixth format is used, each element of the first operand array is multiplied by the corresponding element of the second source operand, and the result is placed in the corresponding element of the destination array.

Rounding takes place when the intermediate result is moved to the destination.

nformat

label **`noformat`** *numvar1 prep dcon1 prep dcon2*
label **`noformat`** *numvar1 prep dcon1 prep numvar3*
label **`noformat`** *numvar1 prep numvar2 prep dcon2*
label **`noformat`** *numvar1 prep numvar2 prep numvar3*
label **`noformat`** *numarray prep dcon1 prep dcon2*
label **`noformat`** *numarray prep dcon1 prep numvar3*
label **`noformat`** *numarray prep numvar2 prep dcon2*
label **`noformat`** *numarray prep numvar2 prep numvar3*

numvar1 is the destination operand
numarray is the destination operand
dcon1 is the new left of decimal point size
dcon2 is the new right of decimal point size
numvar2 contains the new left of decimal point size
numvar3 contains the new right of decimal point size

Flags affected: over

The nformat statement changes the physical length of a numeric variable or of all elements in a numeric array.

The size of the destination numeric variable is changed to **`L.R`** where **L** is the new left of decimal point size and **R** is the new right of decimal point size. If a numeric variable is used to specify the size, the value is obtained by truncation, not rounding.

The new total size of the variable cannot be greater than the original declared total size of the variable when compiled. If the new size would be greater than the declared size, the over flag is set and no changes occur.

After the destination variable has been sized, it is set to zero.

noeject

label **`noeject`**

Flags affected: none

The noeject statement inhibits inclusion of the trailing form feeds in print spool files.

noreturn

label `noreturn`

Flags affected: over

See also: call, return

If the return stack is not empty, noreturn discards the return address on the top of the return stack and sets the over flag to false. If the return stack is empty, the over flag is set to true. In both cases, execution continues with the next statement. If a make or destroy routine is currently executing and the return address to exit the class method is on the top of the return stack, the noreturn statement is ignored and the over flag is set.

not

label **`not`** *charexp prep charvar*
label **`not`** *hexcon prep charvar*
label **`not`** *numexp prep numvar*

charexp is the source operand
hexcon is the source operand
numexp is the source operand
charvar is the destination operand
numvar is the destination operand

Flags affected: equal, eos

For the first and second formats of the not statement, the bitwise not operation is performed on the form pointed character of the source operand. The result is stored in the form pointed position of the destination operand. If a decimal, hexadecimal, or octal constant is used as an operand, the character represented by that character code is used.

If the result is binary zero, the equal flag is set. If either string is null, the eos flag is set and no changes are made.

For the third format of the not statement, the source operand is converted to a 32 bit integer and the not operation is performed on that value. The result is moved to the destination operand. If the result is zero, equal is set. Otherwise, equal is cleared.

The result of the not operation is determined by manipulating the bits in the source operand:

NOT 0 evaluates to 1
NOT 1 evaluates to 0

open

label **`open`** *file* , *charexp1*
label **`open`** *file* , *charexp1, list*
label **`open`** *ifile* , *charexp1*
label **`open`** *ifile* , *charexp1, list*
label **`open`** *afile* , *charexp1*
label **`open`** *afile* , *charexp1, list*
label **`open`** *resource* , *charexp2*
label **`open`** *device* , *charexp2*

resource is the resource variable
device is the device variable
charexp1 is the variable, literal, or character expression that contains the file name
charexp2 is the name operand
file is the label of a file declaration
ifile is the label of an ifile declaration
afile is the label of an afile declaration
list is the comma delimited list of open options

Flags affected: none

See also: close

In the first six forms, the open statement logically connects an operating system file or files with the file, ifile, or afile variable in a DB/C program.

If the second operand is a character variable, the logical string is used. The file name in the second operand must be a standard DB/C file name or a valid file name specific to the operating system. If an extension is not specified, **.txt** is assumed for a file declaration, **.isi** is assumed for an ifile declaration, and **.aim** is assumed for an afile declaration. If the file does not exist or can not be accessed because of security considerations, an error occurs. To open a file, ifile, or afile that is already open, an implicit close is done before the open.

In the last two forms of the open statement, the resource or device specified by the name operand is logically connected to the resource or device variable.

The operand list may contain one of these sharing modes: **share**, **read**, or **exclusive**. If none is specified, **share** is the default.

Any number of programs may have a file opened in share mode. An error will occur if a share mode open is attempted when the file is already opened in read or exclusive mode. In share mode, no input or output buffering is done. share mode performance is slower than read or exclusive performance.

Any number of programs may have a file open in read mode. An error will occur if a read mode open is attempted when the file is already opened in share or exclusive mode. No file writes or updates are allowed in read mode. File input is buffered, so performance will usually be better with read mode input than with share mode input.

Only one program may have a file open in exclusive mode. File input and output is buffered, so performance is faster than for files opened in share or read modes.

The operand list may contain one of these record lock options: **lockmanual** or **lockauto**. If neither is specified, **lockmanual** is the default. If lockauto is in effect, then each read statement is implicitly converted to a locked read statement. If lockmanual is in effect, then the record locking form of the read statement must be specified to lock a record (for example: readlk).

The operand list may contain one of these record lock options: **multiple** or **single**. If neither is specified, **multiple** is the default. If single is in effect, then any attempt to lock a second record through this file, ifile, or afile variable will cause the previously locked record to be unlocked. If multiple is in effect, then any number of records may be locked concurrently.

The operand list may contain one of these record lock options: **wait** or **nowait**. If neither is specified, **wait** is the default. If nowait is in effect, then any attempt to lock a record that is already locked will cause the read statement to return immediately and to set the less flag. If wait is in effect, all locking read statements will wait for a record to be avail able before returning.

The operand list may contain a character variable or literal if the open is for an afile variable. This option specifies the alternate match character. The first character in the logical string is the new match character.

The alternate match character may also be specified by the option **match=***charvar* or **match=***charlit*. The form pointed character of the variable or literal specifies the new match character.

If none of these options are specified or if the logical string is null, then the match character is the character that was specified in the aimdex parameter list. If the match character was not specified there, it defaults to a question mark (?).

If the operating system has no available file handles or the **dbcdx.file.openlimit** runtime property value would be exceeded, then a non-exclusively opened file that does not have an active filepi or record lock will be physically closed. The file will remain logically open and will be physically reopened on the first I/O instruction for the corresponding file declarations. Exceeding either the operating system limit or the **dbcdx.file.openlimit** could allow another user to open the physically closed file in a non-compatible mode.

or

label **`or`** *charexp prep charvar*
label **`or`** *hexcon prep charvar*
label **`or`** *numexp prep numvar*

charexp is the source operand
numexp is the source operand
charvar is the destination operand
numvar is the destination operand

Flags affected: equal, eos

See also: and, not, xor, rotate

For the first and second formats of the or statement, the bitwise or operation is performed on the form pointed characters of the source and destination operands. The result is stored in the form pointed position of the destination operand. If a decimal, hexadecimal, or octal constant is used as an operand, the character represented by that character code is used. If the result is binary zero, the equal flag is set. If either string is null, the eos flag is set and no changes are made.

For the third format of the or statement, the source and destination operands are converted to 32 bit integers and the or operation is performed. The result is moved to the destination operand. If the result is zero, equal is set. Otherwise, equal is cleared.

The result of the or operation is determined by comparing the bits in each operand:

0 OR 0 evaluates to 0
0 OR 1 evaluates to 1
1 OR 0 evaluates to 1
1 OR 1 evaluates to 1

pack

label **`pack`** *charvar prep list*

charvar is the destination operand
list is a list of character and numeric variables, literals, arrays, and list variables

Flags affected: eos

The pack statement combines two or more strings into a single variable. The logical strings of each entry in the list are appended together and moved to the destination operand. If the resulting string is too large to fit into the destination operand, the string is truncated and the eos flag is set.

If the form pointer of a variable in the list is zero, that variable is ignored. If every variable in the list has a form pointer with a value of zero, then the form pointer and the logical length pointer of the destination are also set to zero. Otherwise, the form pointer of the destination variable is set to one and the length pointer is set to point to the last character moved. If the destination operand also appears in the list of source variables, the results are undefined.

packlen

label **`packlen`** *charvar prep list*

charvar is the destination operand
list is a list of character and numeric variables, literals, arrays, and list variables

Flags affected: eos

The packlen statement combines two or more variables or literals into a single variable. The values of each of the variables in the list is appended together and stored in the destination, including variables nested inside list variables and all elements of arrays. If the resulting string is too large to fit into the destination operand, the string is truncated and the eos flag is set.

All characters from literals and numeric variables are appended.

For character variables, characters are appended starting with the first character in the variable through the character pointed to by the length pointer. Blanks are appended for each character position after the length pointer through the maximum length of the variable. If the form pointer of the variable is zero, then blanks are appended for each character position through the maximum length of the variable.

The form pointer of the destination operand is set to one and the length pointer is set to point to the last character moved. If the destination operand also appears in the list of source variables, the results are undefined.

pause

label **`pause`** *numexp*

numexp is the length of time in seconds

Flags affected: none

The pause statement suspends execution of a program for the number of seconds specified in the operand. The value of *numexp* may be fractional.

perform

label **`perform`** *numvar prep list*

numvar is the index
list is a comma delimited list of program execution labels and program label variables

Flags affected: none

See also: call, return, branch

The perform statement causes program execution to continue with one of the entries in list based on the value of the index. Program execution continues at the statement specified by the N^{th} label in the list. N is determined by converting the index to an integer value.

The specified label must be part of a subroutine so that a return can be made to the statement which follows the perform instruction. The address of the statement following the perform statement is put on the return stack.

If the index is fractional, the fraction is truncated, not rounded. If the resulting integer is less than one or greater than the number of entries in the list, then no perform takes place and execution continues with the statement immediately following the perform instruction.

ploadmod

label **`ploadmod`** *charexp*

charexp is the source operand

Flags affected: none

See also: loadmod

The ploadmod statement loads a file as a secondary program module. If the source operand is a character variable, the logical string is used as the file name. If the file is not found or is otherwise invalid, an error occurs.

The ploadmod statement functions in the same way as loadmod except that the module loaded by ploadmod is permanently loaded. The ploadmod statement performs the same action as the **dbcdx.preload** runtime property.

popreturn

label **`popreturn`** *lblvar*

lblvar is the destination label variable

Flags affected: over

See also: call, noreturn, pushreturn, return

The popreturn statement causes the value of a program label to be removed from the top of the return stack and to be stored in the destination label variable. This decreases the number of entries in the return stack by one.

If there are no entries on the return stack before this statement executes, the over flag is set and the destination label variable is cleared. If a make or destroy routine is currently executing and the return address to exit the class method is on the top of the return stack, the popreturn statement clears the destination label variable and the over flag is set. Otherwise, the over flag is cleared.

prepare

label **`prepare`** *file* `,` *charexp1*
label **`prepare`** *file* `,` *charexp1* `,` *mode*
label **`prepare`** *file* `,` *charexp1* `,` *charexp2*
label **`prepare`** *ifile* `,` *charexp1*
label **`prepare`** *ifile* `,` *charexp1* `,` *charexp2*
label **`prepare`** *afile* `,` *charexp1* `,` *charexp2* `,` *key-info*
label **`prepare`** *afile* `,` *charexp1* `,` *key-info*
label **`prepare`** *device* `,` *charexp3*
label **`prepare`** *resource* `,` *charexp3*

`prep` may be used in place of **`prepare`**

file is the label of a file declaration
ifile is the label of an ifile declaration
charexp1 is the variable, literal, or expression that contains the file name
mode is the prepare mode
charexp2 is the variable, literal, or expression that contains the data file name
key-info is a list of parameters that specify the key information
device is the device variable
resource is the resource variable
charexp3 contains the keywords and parameters

Flags affected: none

See also: open

In the first seven forms, the prepare statement is used to create a new file or to erase the contents of an existing file. The prepare statement logically connects an operating system file or files with the file, ifile, or afile variable in the DB/C program.

The prepare statement causes the file or files to be opened in exclusive mode. The same considerations apply as with an open operation in exclusive mode.

In the second form, the mode must be either **create** or **prepare**. **create** mode causes an error to occur when an attempt is made to prepare a file that already exists. If the mode is prepare, or if any of the other six forms is used, then an empty file(s) is created if it does not exist and is made empty if it does exist.

If the third form is specified, and the file type is native the logical string of *charexp2* is passed as the second argument to the nioprep function. If the file type is not native, *charexp2* is ignored.

If the first, second, or third form is used, only a data file is prepared. The logical string of *charexp1* specifies the data file name. If an extension is not specified, **.txt** is assumed.

If the fourth or fifth form is specified, a data file and an index file are prepared. The logical string of *charexp1* specifies the index file name. If an extension is not specified, **.isi** is assumed. If *charexp2* is not specified, then the data file name defaults to the index file name with an extension of **.txt**. If *charexp2* is specified, then its logical string is the data file name. If the extension is not specified, the default is **.txt**.

If the sixth or seventh form is specified, a data file and an associative index file are prepared. The logical string of *charexp1* specifies the associative index file name. If an extension is not specified, **.aim** is assumed. If *charexp2* is not specified, then the data file name defaults to the associative index file name with an extension of **.txt**. If *charexp2* is specified, then its logical string is the data file name. If the extension is not specified, the default is **.txt**.

The *key-info* may include these parameters: **keys=***charexp*, **match=***charexp*, and/or **case=***charexp*.

The **keys=***charexp* parameter is required. The logical string of the keys parameter specifies the key information. The key information consists of one or more comma delimited key position specifications. A key position specification is either a single positive number or two positive numbers separated by a hyphen. If the two-number form is used, the first number must be less than or equal to the second number. A key position specification defines the character position(s) of each of the associative index keys within a record. The first key position specification corresponds to the first key; the second key position specification corresponds to the second key, etc. The key position specification can be immediately preceded by a letter X which indicates an excluded key field. An excluded key field is used to match key data and record fields, but is not used to generate key information for the associative index. The key information must have at least one key position specification which is not excluded.

The form pointed character of the **case=***charexp* parameter specifies the key type specification. The key type specification may be either a letter **u** representing case insensitive or a letter s representing case sensitive. If the case parameter is not specified, **u** is the default.

The form pointed character of the **match=***charexp* parameter specifies the universal match character. If the match parameter is not specified, the universal match character defaults to the question mark character (**?**).

In the last two forms, the prepare statement creates and opens a device or resource.

If the operating system has no available file handles or the **dbcdx.file.openlimit** runtime property would be exceeded, then a non-exclusively opened file that does not have an active filepi or record lock will be physically closed. The file will remain logically open and will be physically reopened on the first I/O instruction for the corresponding file declarations. Exceeding either the operating system limit or the **dbcdx.file.openlimit** runtime property could allow another user to open the physically closed file in a non-compatible mode.

print

label **`print`** *list*
label **`print`** *pfile* ; *list*

list is a comma delimited list of character variables, numeric variables, image variables, list variables, arrays, literals, octal or hexadecimal constants, and print control codes

pfile is the label of a pfile declaration

Flags affected: none

See also: splopen, splopt, splclose, format

The print statement prints or spools data. The operand list consists of variables, literals, and ***** control codes.

The print position defines where in the print line the characters are printed on the printer or in the spool file. The print position starts at position one (the far left character position) for each line. After variables and literals are printed, the new print position is immediately to the right of the last character printed.

When a character variable is in the operand list, characters are printed from the first physical character through the character pointed to by the length pointer. A blank is then printed for each character position after the length pointer through the maximum length of the variable. If the form pointer of the variable is zero, the field is considered cleared and only blanks are printed up to the maximum length of the string.

When a numeric variable or literal is in the operand list, all characters are printed starting with the first character.

When an image variable is in the operand list, its contents are printed when using the graphical version of DB/C DX. The image variable is ignored in the non-graphical versions unless one of the following SPLOPEN options was specified: **L=PS**, **L=PCL**, or **L=PDF**. Only one colorbit images are supported when using **L=PCL**.

f** or ***f=n* is the form feed control code. The ***f** control code causes the printer to skip to top of the next page and the print position to be set to one. *n* is an equate, decimal constant, or numeric variable that represents the number of pages to feed. If *n* is equal to zero, the control code is ignored.

c** or ***c=n* is the carriage return control code. This control code causes the print position to be set to one in the current line. *n* is an equate, decimal constant, or numeric variable. If *n* is equal to zero, the control code is ignored. Otherwise, *n* has no effect on the functioning of the control code.

l** or ***l=n* is the line feed control code. The ***l** control code causes the printer to advance to the next line. The print position is unchanged. *n* is an equate, decimal constant, or numeric variable that represents the number of lines that the printer will advance. If n is equal to zero, the control code is ignored.

n** or ***n=n* is the next line control codes. The ******n* control code causes the printer to ad vance to the beginning of the next line. *n* is an equate, decimal constant, or numeric variable that represents the number of lines to advance. If *n* is not specified, it defaults to one. If *n* is equal to zero, the control code is ignored.

***zf** is the zero fill control code. This control code applies only to the next variable in the operand list. If the next variable is a numeric variable, then blank characters in the variable are printed as zeros. Additionally, if the variable is negative, the minus sign is printed in the first character position.

***zs** is the zero suppress control code. This control code affects only the next variable in the operand list. If the next variable is a numeric variable with a zero value, blanks are printed for each character position of the numeric variable (including the decimal point). Otherwise, the variable is printed normally.

sl** (or **+) is the blank suppression control code. This control code affects all remaining character variables in the operand list. This control code causes no blanks to be printed for the characters between the length pointer and the maximum length of the string. If the variable is cleared, then nothing is printed and the print position remains unchanged.

***ll** is the logical string print control code. This control code affects printing of all remaining character variables in the operand list. This control code causes the characters contained in the logical string of the variable to be printed. If the form pointer of the variable is zero, no characters are printed and the print position is not changed.

***pl** (or ***-**) is the print suppression off control code. This control code causes printing of character variables to revert to the normal method of printing. It cancels the effect of the ***sl** and ***ll** control codes.

format=charexp* is the format control code. This control code affects the next variable in the operand list. *charexp* is a formatting mask which is used to reformat numeric and character data. Refer to the explanation of the format statement for information about how the mask is used.

rptchar=charexp* **:** *n* is the repeat character control code. The equal sign may be replaced by a colon or a blank space. *charexp* is a character variable, character literal, or character expression. If *charexp* is a variable, then the form pointed character is the character to be repeated. *n* is a decimal constant or numeric variable that specifies the number of times the character is to be repeated. Line wrap does not occur; that is, the character is only repeated to the end of the current line. The print position is set to the position after the last character printed.

tab=n* or ******n* are the tab control codes. These control codes adjust the print position within the current print line. *n* is an integer decimal constant or numeric variable that designates the new print position. The value of *n* may be 1 through 400, inclusive. The value of the numeric variable is truncated if necessary.

***flush** is the flush print buffer control code. This control code causes pending print lines to be output immediately.

font=charexp* is the font control code. *charexp* is a character variable, character literal, or character expression that represents the name of a font. This control code affects all variables and literals that follow in the operand list until another ***font** control code is encountered. All characters affected by a particular ***font** control code will be printed in the specified font. This control code is ignored for print destinations that do not support multiple fonts.

color=n* is the color control code. *n* is a numeric variable, numeric expression, or fixed color code which specifies the new print color. This value is an integral value that corresponds to a 24-bit RGB value (red is low eight bits, green is middle eight bits, and blue is high eight bits). The default is black. This control code is ignored for print destinations that do not support multiple colors. Fixed color codes are: ***black**, ***blue**, ***green**, ***cyan**, ***red**, ***magenta**, ***yellow**, and ***white**.

p=h* **:** *v* is the pixel print position control code. *h* is the horizontal position and *v* is the vertical position in pixels. The upper left corner of the page is 1:1. This control code is useful for fine control of the print position. The relationship between character print position and pixel print position is undefined, although they both manipulate the single print position. This control code is ignored for print destinations that do not support pixel positioning.

line=h* **:** *v* is the print line control code. *h* and *v* may be decimal constants or numeric variables. A line is drawn from the current print position to the position specified by *h* and *v*. The current print position is set to *h* and *v*. This control code is ignored for print destinations that do not support pixel positioning and line drawing.

linewidth=n* is the specify linewidth control code. *n* may be a decimal constant or numeric variable. It specifies the width, in pixels, of lines printed with the ***line** control code.

***rj** is the right justify control code. This control code applies only to the next variable or literal in the operand list. The characters of the next variable or literal flow left from current print position. The print position remains unchanged. This control code is ignored for print destinations that do not support pixel positioning.

cj=n* is the center justify control code. This control code applies only to the next variable or literal in the operand list. The characters of the next variable or literal are centered in an area starting with the current print position that is the number of pixels wide that is specified by the value *n*. If the width of the text is too wide for the area, this control code is ignored and the text is printed normally. Otherwise, the resulting print position is the old starting position incremented by the width of the area, not the width of the text printed.

textangle=n* is the print angled text control code. This control code applies only to the next variable or literal in the operand list. The characters of the next variable or literal are printed such that the upper left corner of the text is at the current print position. *n* defines the angle, in degrees, at which the text is printed. The angle is the same as for a compass, that is north (up) is zero degrees, east (normal) is 90 degrees, south (down) is 180 degrees, and west (upside down) is 270 degrees.

fb=charexp* is a combination of a form feed and a request to change the input paper bin. This control code applies only to the Windows cooked output type or the PCL output format. For cooked output *charexp* is a bin name recognized by the driver. Use the getpaperbins statement to find out which bin names the printer driver recognizes. For PCL output, only the first character of *charexp* is used and it should be a number. A PCL escape sequence is built using this number and its meaning will depend on the particular printer that is used.

rectangle=h***:***v* is the print a filled rectangle control code. The current draw position is one corner of the rectangle, *h***:***v* specifies the diagonally opposite corner of the rectangle. *h* and *v* may be decimal constants or numeric variables. The filled rectangle is printed in the current color. The current draw position is not changed.

box=h***:***v* is the print a box control code. The current draw position is one corner of the box, *h***:***v* specifies the diagonally opposite corner of the box. The box is printed with a line that is the current line width and type. The box is printed in the current color. The current draw position is not changed.

circle=n* is the print a circle control code. *n* may be a decimal constant or numeric variable. A circle is printed in the current color with the center at the current position and a radius of *n* pixels. The circle is printed with a line that is the current line width and type. The current draw position is not changed.

bigdot=n* is the print a filled circle control code. *n* may be a decimal constant or numeric variable. A filled circle is printed in the current color with the center at the current position and a radius of *n* pixels. The current draw position is not changed.

triangle=h1***:***v1***:***h2***:***v2* is the print a filled triangle control code. *h1*, *v1*, *h2*, and *v2* may be decimal constants or numeric variables. The current position is one corner of the triangle. the parameters are the coordinates of the other two vertices. The filled triangle is printed in the current color. The current draw position is not changed.

pushreturn

label **`pushreturn`** *prog-label*
label **`pushreturn`** *lblvar*

prog-label is the source program label
lblvar is the source label variable

Flags affected: none

See also: call, noreturn, popreturn, return

The pushreturn statement causes the value of the source operand to be put onto the top of the return stack, thus increasing the number of entries in the return stack by one.

If the source label variable contains an invalid value, an E 551 error occurs. If the return stack is full before this statement executes, an E 501 error occurs.

put

label `put` *queue*`;` *list*

queue is the queue variable
list is the list of character and numeric variables, literals, and control codes

Flags affected: over

The put statement moves a message from the list to the end of the queue specified by the queue variable. If the queue is full before the message is moved to the queue, then the oldest message is discarded, the message is moved to the queue, and the over flag is set.

***sl** is the blank suppression control code. This control code affects all remaining character variables in the list. This control code causes no blanks to be sent for the characters between the length pointer and the maximum length of the string. If the variable is cleared, then no characters are sent.

***ll** is the logical length control code. This control code is in effect for all character variables remaining in the list or until a ***pl** control code is encountered. The logical string of the character variable is sent.

***pl** is the physical length control code. This control code cancels the effect of the ***ll** and ***sl** control code.

putfirst

label **`putfirst`** *queue* ; *list*

queue is the queue variable
list is the list of character and numeric variables, literals, and control codes

Flags affected: over

The putfirst statement moves a message from the list to the beginning of the queue specified by the queue variable. If the queue is full before the message is moved to the queue, then the oldest message is discarded, the message is moved to the queue, and the over flag is set.

***sl** is the blank suppression control code. This control code affects all remaining character variables in the list. This control code causes no blanks to be sent for the characters between the length pointer and the maximum length of the string. If the variable is cleared, then no characters are sent.

***ll** is the logical length control code. This control code is in effect for all character variables remaining in the list or until a ***pl** control code is encountered. The logical string of the character variable is sent.

***pl** is the physical length control code. This control code cancels the effect of the ***ll** and ***sl** control code.

query

label **`query`** *resource*`,` *charexp*`;` *list*
label **`query`** *device*`,` *charexp*`;` *list*

resource is the resource variable
device is the device variable
charexp is the function operand
list is a list of character and numeric variables

Flags affected: none

See also: change, open

The query statement causes the logical string of the function operand to be sent to the specified resource or device. The resource or device responds by filling the variables in the list.

read

label **`read`** *file* `,` *numvar* `;` *list*
label **`read`** *ifile* `,` *var* `;` *list*
label **`read`** *afile* `,` *numvar* `;` *list*
label **`read`** *afile* `,` *key-list* `;` *list*

`readtab` may be used in place of **`read`**

file is the label of a file declaration
ifile is the label of an ifile declaration
afile is the label of an afile declaration
numvar is the second operand
var is the second operand
list is the list of variables and tab control codes
key-list is a list of character variables, character arrays, and list variables

Flags affected: less, over

See also: open, readkg, readlk

The read statement reads data from a file and places it into the variables in the list.

The read is random if the second operand is a non-negative numeric variable. The read is sequential if the second operand is a negative numeric variable. The read is indexed if the first operand is an ifile and the second operand is a character variable. The read is aimdexed if the first operand is an afile and the second operand is a character variable.

Characters are read from the file and placed in the variables in the list. The way in which data is moved to each variable depends on the type of variable.

If the variable in the list is a character variable, each character is moved from file into the variable starting with the first physical character position in the variable and continuing through the maximum length of the variable. If the end of the logical record is encountered before the variable is full, then all characters are stored. The form pointer is set to one and the logical length pointer is set to the last character moved. The form pointers of all remaining character variables are set to zero after the end of record is reached.

If the variable in the list is a numeric variable, the characters moved from file must be valid numbers with the same format as the numeric variable. That is, the decimal point (if present) must be in the same position in the numeric data as in the variable. If the decimal point is aligned and the characters constitute a valid number, then the characters are moved to the numeric variable. If the format of the characters in the file was created using the ***mp** control code, then the values are first converted to valid numeric format and then moved to the numeric variable. Zero is moved to any numeric variables left in the list after the logical end of record is reached. If the characters do not constitute a valid number or if the logical end of record occurs while filling a numeric variable, then a format error occurs.

If the list is terminated with a semicolon (`;`), then at the end of the read operation, the file position points to the next character in the current logical record. This is considered to be a partial read. If the list is not terminated with a semicolon, then at the end of the read operation, the file position points to the character after the end-of-record character in the current logical record.

If the first character to be moved into the first variable of the list is an end-of-file character, then over is set. Additionally, the form pointers of all character variables are set to zero and zero is moved to each numeric variable. No change is made to the file position.

The list may contain tab control codes. A tab control is specified by ***tab=***n* or ******n* , where *n* is an integer decimal constant or numeric variable. Values of *n* up to 65500 are allowed.

If the read operation is for a file, ifile, or afile variable that was opened with the lockauto option, then the read operation is implicitly converted to a readlk operation. less is affected only in this case.

If the access is sequential and the second operand has a value of -3, then the file is positioned to the end of file and the read takes place like a normal read that encountered end of file. If the access is sequential and the second operand has a value of -4, then the file is read backwards sequentially, beginning with the record before the last record read. If the access is sequential and the second operand has a value other than -3 or -4, then the file is read sequentially starting with the character pointed to by the current file position.

If the access is random, then the Nth record of the file is read where N is the truncated value of the numeric variable that is the second operand. If the record number (N) is beyond the end of file, then a range error occurs. The first record in the file has a record number value of zero.

If the access is indexed, the logical string from the character variable that is the second operand is the lookup key used to retrieve the logical record. The lookup key matches the key in the index if they contain identical characters and have the same number of characters. The lookup key also matches the key in the index if the lookup key has more characters than the index key, all characters in the index key match characters in the lookup key, and all remaining characters in the lookup key are blank. The lookup key also matches the key in the index if the lookup key has fewer characters than the index key, all characters in the lookup key match characters in the index key, and all remaining characters in the index key are blank. If none of these cases occurs, the lookup key does not match a key in the index.

If the lookup key matches a key in the index, then the record is read and the variables in the list are filled normally. If the lookup key does not match a key in the index, then over is set and the variables are cleared and zeroed. If the form pointer of the second operand is zero, the lookup key is considered null, and the last accessed record is reread. If the last access to the index file was unsuccessful, then an error occurs.

If the access is aimdexed, then logical strings from each character variable in the key-list are used to construct the read match pattern. If one or more records in the file match the read match pattern, then one of those records is read into the variable list just as with any other read access. The remaining records (if any) that matched the read match pattern may be read with the readkg operation. If no records in the file match the read match pattern, over is set and the variables are cleared and zeroed. If all the logical strings in the key-list are null, then the previously read record is reread without destroying or changing any readkg information.

To be a valid key, the first two characters of the logical string must be valid field numbers, the third character (the search type) must be one of X, L, R, or F, and the remaining characters are the search key characters. A logical string of a variable in the key-list must be of length four or greater. If the logical string is null, it is ignored. If it is not null and has an invalid format, an error occurs. The field numbers correspond with those specified in the aimdex parameter list. Only records that match all the key specifications are considered to match.

An X-type search specifies that an exact match must occur between the characters in the key and the characters in the field in the record. If the length of the field is less than the number of characters from the key, then the key string is truncated. If the length of the field is greater than the number of characters from the key, then the key string is filled with blanks to the right.

An L-type search specifies that a left side of field match must occur between the characters in the key and the characters in the field in the record. If the length of the field is less than or equal to the number of characters from the key, then the key specification is treated as if it were an X-type search key.

An R-type search specifies that a right side of field match must occur between the characters in the key and the characters in the field in the record. If the length of the field is less than or equal to the number of characters from the key, then the key specification is treated as if it were an X-type search key.

An F-type search specifies that a floating match must occur between characters in the key and characters in the field in the record. If the length of the field is less than or equal to the number of characters in the key, then the key specification is treated as if it were an X-type search key.

The current match character, as specified in the open statement or elsewhere, modifies the X, L, R, and F matching process. The current match character is a wild card character that matches any character from the field in the record.

Aimdexed access is based on keys that are left justified, right justified, and floating. At least one key in the key-list must meet one of the following criteria:

1. An X-type search key that has at least one non-blank, non-wild card character as the far right character or the far left character,

2. An L-type search key that has a non-blank, non-wild card character as the far left character,
3. An R-type search key that has at least one non-blank, non-wild card character as the far right character or the far right character,
4. An F-type search key that has at least three contiguous non-blank, non-wild card characters.

If one of these requirements is not met, an error occurs. In general, the more keys of this nature that are specified, the better is the performance of the read access.

readgplk

label **`readgplk`** *afile* `;` *list*

afile is the label of an afile declaration
list is the list of variables and tab control codes

Flags affected: less, over

See also: read, readkgp, open

The readgplk statement is the same as the readkgp statement, except that the accessed record is locked.

If the record is already locked by another program and the wait lock option was specified in the open statement, then the readgplk operation will wait (possibly forever) for the record to be unlocked by the other program.

If the record is already locked by another program and the nowait lock option was specified in the open statement, then the readgplk operation will return immediately, the variables in the list will be zeroed and cleared, the less flag will be set, and the over flag will be cleared.

If the record is successfully read and locked, the less and over flags are cleared.

The list may contain tab control codes. A tab control is specified by **`*tab=`***n* or **`*`***n* , where *n* is an integer decimal constant or numeric variable. Values of *n* up to 65500 are allowed.

readkg

label **`readkg`** *afile* `;` *list*

`readkgtb` may be used in place of **`readkg`**

afile is the label of an afile declaration
list is the list of variables and tab control codes

Flags affected: less, over

See also: open, read

The readkg statement reads the next record in the aimdexed read match list. If there are no more records that satisfy the last read match pattern, then the over flag is set and the variables in the list are cleared and zeroed. If no previous aimdexed read has occurred, or if there has been an aimdexed write or insert on the same aim file more recently than the last aimdexed read, an error results.

If the readkg operation is for an afile variable that was opened with the lockauto option, then the readkg operation is implicitly converted to a readkglk statement. The less flag is affected only in this case.

The list may contain tab control codes. A tab control is specified by **`*tab=`***n* or **`*`***n* , where *n* is an integer decimal constant or numeric variable. Values of *n* up to 65500 are allowed.

readkglk

label **`readkglk`** *afile* `;` *list*

afile is the label of an afile declaration
list is the list of variables and tab control codes

Flags affected: less, over

See also: open, read, readkg

The readkglk statement functions the same as the readkg statement, except that the accessed record is locked.

If the record is already locked by another program and the wait lock option was specified in the open statement, then the readkglk operation will wait (possibly forever) for the record to be unlocked by the other program.

If the record is already locked by another program and the nowait lock option was specified in the open statement, then the readkglk operation will return immediately, the variables in the list will be zeroed and cleared, the less flag will be set, and over will be cleared.

If the record is successfully read and locked, the less and over flags are cleared.

The list may contain tab control codes. A tab control is specified by **`*tab=`***n* or **`*`***n* , where *n* is an integer decimal constant or numeric variable. Values of *n* up to 65500 are allowed.

readkgp

label **`readkgp`** *afile* `;` *list*

afile is the label of an afile declaration
list is the list of variables and tab control codes

Flags affected: less, over

See also: open, read, readkg

The readkgp statement reads the previous record in the aimdexed read match list. If there are no previous records that satisfy the last read match pattern, then over is set and the variables in the list are cleared and zeroed. If no previous aimdexed read has occurred, or if there has been an aimdexed write or insert on the same aim file more recently than the last aimdexed read, an error results.

If the readkgp operation is for an afile variable that was opened with the lockauto option, then the readkgp operation is implicitly converted to a readgplk operation. The less flag is affected only in this case.

The list may contain tab control codes. A tab control is specified by **`*tab=`***n* or **`*`***n* , where *n* is an integer decimal constant or numeric variable. Values of *n* up to 65500 are allowed.

readkp

label **`readkp`** *ifile* `;` *list*

`readkptb` may be used in place of **`readkp`**

ifile is the label of an ifile declaration
list is the list of variables and tab control codes

Flags affected: less, over

See also: open, read, readks

The readkp statement reads the record associated with the previous logical sequential key. The variables in the list are filled in the same manner as for the read statement.

The previous logical sequential key is the next descending key in the collating sequence of the character set. The last access to the index defines which key is next. Operations such as read, write, insert, readks, readkp, delete, deletek, and the respective tabbed operations allow access to the index. If there is not a prior key (that is, the beginning of the index is reached), the over flag is set and the variables in the list are cleared and zeroed.

If the readkp operation is for an ifile variable that was opened with the lockauto option, then the readkp operation is implicitly converted to a readkslk operation. The less flag is affected only in this case.

The list may contain tab control codes. A tab control is specified by **`*tab`**=*n* or **`*`***n* , where *n* is an integer decimal constant or numeric variable. Values of *n* up to 65500 are allowed.

readkplk

label **`readkplk`** *ifile* `;` *list*

ifile is the label of an ifile declaration
list is the list of variables and tab control codes

Flags affected: less, over

See also: open, read, readkp

The readkplk statement is the same as the readkp statement, except that the accessed record is locked.

If the record is already locked by another program and the wait lock option was specified in the open statement, then the readkplk operation will wait (possibly forever) for the record to be unlocked by the other program.

If the record is already locked by another program and the nowait lock option was specified in the open statement, then the readkplk operation will return immediately, the variables in the list will be zeroed and cleared, the less flag will be set, and the over flag will be cleared.

If the record is successfully read and locked, the less and over flags are cleared.

The list may contain tab control codes. A tab control is specified by **`*tab=`***n* or **`*`***n* , where *n* is an integer decimal constant or numeric variable. Values of *n* up to 65500 are allowed.

readks

label **`readks`** *ifile* **`;`** *list*

`readkstb` may be used in place of **`readks`**

ifile is the label of an ifile declaration
list is the list of variables and tab control codes

Flags affected: less, over

See also: open, read, readkp

The readks statement reads the record associated with the next logical sequential key. The variables in the list are filled in the same manner as for the read statement.

The next logical sequential key is the next ascending key in the collating sequence of the character set. The last access to the index defines which key is next. Operations such as read, write, insert, readks, readkp, delete, deletek, and the respective tabbed operations allow access to the index. If this is the first access, the record associated with the first key in the collating sequence is read. If there are no more keys after the last one accessed, the over flag is set and the variables in the list are cleared and zeroed.

If the readks operation is for an ifile variable that was opened with the lockauto option, then the readks operation is implicitly converted to a readkslk operation. The less flag is affected only in this case.

The list may contain tab control codes. A tab control is specified by **`*tab=`***n* or **`*`***n* , where *n* is an integer decimal constant or numeric variable. Values of *n* up to 65500 are allowed.

readkslk

label **`readkslk`** *ifile* `;` *list*

ifile is the label of an ifile declaration
list is the list of variables and tab control codes

Flags affected: less, over

See also: open, read, readks

The readkslk statement is the same as the readks statement, except that the accessed record is locked.

If the record is already locked by another program and the wait lock option was specified in the open statement, then the readkslk operation will wait (possibly forever) for the record to be unlocked by the other program.

If the record is already locked by another program and the nowait lock option was specified in the open statement, then the readkslk operation will return immediately, the variables in the list will be zeroed and cleared, the less flag will be set, and the over flag will be cleared.

If the record is successfully read and locked, the less and over flags are cleared.

The list may contain tab control codes. A tab control is specified by **`*tab=`***n* or **`*`***n* , where *n* is an integer decimal constant or numeric variable. Values of *n* up to 65500 are allowed.

readlk

label **`readlk`** *file* `,` *numvar* `;` *list*
label **`readlk`** *ifile* `,` *var* `;` *list*
label **`readlk`** *afile* `,` *numvar* `;` *list*
label **`readlk`** *afile* `,` *key-list* `;` *list*

file is the label of a file declaration
ifile is the label of an ifile declaration
afile is the label of an afile declaration
numvar is the second operand
var is the second operand
list is the list of variables and tab control codes
key-list is a list of character variables

Flags affected: less, over

See also: open, read

The readlk statement is the same as the read statement, except that the accessed record is locked.

If the record is already locked by another program and the wait lock option was specified in the open statement, then the readlk operation will wait (possibly forever) for the record to be unlocked by the other program.

If the record is already locked by another program and the nowait lock option was specified in the open statement, then the readlk operation will return immediately, the variables in the list will be zeroed and cleared, the less flag will be set, and the over flag will be cleared.

If the record is successfully read and locked, the less and over flags are cleared.

The list may contain tab control codes. A tab control is specified by **`*tab=`***n* or **`*`***n* , where *n* is an integer decimal constant or numeric variable. Values of *n* up to 65500 are allowed.

recv

label **`recv`** *cfile*`,` *numvar*`;` *list*

cfile is the label of a comfile declaration
numvar is the timeout value
list is a list of variables

Flags affected: none

See also: comclr, comtst, comwait, recvclr, send, wait

The recv statement initiates the receipt of data from another task. *list* is the list of variables that are filled with characters received. When the recv statement is executed, the status of the comfile is changed to "receive-pending".

The timeout value is the number of seconds the recv statement will remain "receive-pending" before it is canceled. If the timeout value is -1, then the recv will never timeout. If the timeout value is zero, the recv will timeout immediately (if there is no data to receive).

The receive status of a comfile is set to "receive-clear" before an attempt to receive data is made. Therefore, if the receive status of a comfile is "receive-pending" and a recv statement for the same comfile is executed, the first recv operation will be canceled.

When the recv is complete, the status of the comfile is changed to "receive-complete". The comtst statement must be executed to fill the list of variables with the characters received.

The receiving variables do not have to be the same lengths as the sending variables. For example, if the send list contains two variables of length three, the recv list can contain one variable of length six. If the recv list cannot contain all the data sent, then the excess data is ignored. The form pointers and logical length pointers of any variables in the recv list that did not receive data are set to zero.

recvclr

label **`recvclr`** *cfile*

cfile is the label of a comfile declaration

Flags affected: less

See also: recv

The recvclr statement sets the receive status of the comfile to "receive-clear". It does not affect the send status. If the status prior to execution of the recvclr statement was "receive-pending", then the recv is canceled and less flag is set.

release

label **`release`**

Flags affected: over

See also: splclose

The release statement causes the currently allocated printer device to be deallocated. The release statement is ignored if the current print output is to a file. The release operation does not cause a splclose to occur. If a printer device is not currently allocated then an allocation of the device is attempted. If the device is successfully allocated, it is then deallocated and the over flag is cleared. Other wise, the over flag is set. The behavior of release on a non-allocated device may be operating system dependent.

rename

label **`rename`** *charexp1 prep charexp2*

charexp1 is the old file name
charexp2 is the new file name

Flags affected: none

The rename statement renames a file. The logical string of the *charexp1* operand specifies the old file name. The logical string of the *charexp2* operand specifies the new file name. If the rename is unsuccessful, then an IO error occurs.

replace

label **`replace`** *charexp prep charvar*

`rep` may be used in place of **`replace`**

charexp is the source operand
charvar is the destination operand

Flags affected: eos

Characters in the destination operand are replaced by characters from the source operand.

The logical string of the source operand contains pairs of characters. The first member of each pair is the search character. The second member of each pair is the replacement character. Each character of the logical string of the destination operand is compared to the search characters from each pair of characters in the source operand.

If a search character matches the character in the destination operand, the replacement character of the pair is moved to that character position in the destination operand. Replacements are performed from left to right, one character at a time. All characters from the source are searched before a replacement is made. If the search character is duplicated in the source, the replacement character is the one that follows the last occurrence of the search character in the source.

The form pointers and length pointers of both operands are unchanged. If the length of the source logical string is not an even number, the eos flag is set and no replacements are made.

reposit

label **`reposit`** *file*`,` *numvar*
label **`reposit`** *ifile*`,` *numvar*
label **`reposit`** *afile*`,` *numvar*

file is the label of a file declaration
ifile is the label of an ifile declaration
afile is the label of an afile declaration
numvar is the source operand

Flags affected: equal, over

See also: fposit

The reposit statement changes the value of the current file position to the value of the source operand.

If the value of the source operand is the same as the end-of-file position, then the over flag is set. If the value of the source operand is greater than the end-of-file position, then no change to the file position takes place and the equal flag is set. If the value of the source operand is zero, the file is set to the position of or before the first record.

The reposit instruction is often used with the fposit instruction to save a file position and restore it later.

reset

label `reset` *charvar1*
label `reset` *charvar1 prep dcon*
label `reset` *charvar1 prep numvar*
label `reset` *charvar1 prep charvar2*
label `reset` *charvar1 prep charlit*
label `reset` *charvar1 prep expression*

charvar1 is the destination operand
dcon is the reset value
numvar is the variable containing the reset value
charvar2 is the character variable used to determine the reset value
charlit is the character literal used to determine the reset value
expression is a character or numeric expression. The result of the expression determines the reset value

Flags affected: eos

The reset statement sets the form pointer of the destination to the specified value.

If the first format is used, the value of the destination form pointer is set to one.

If the second format is used, the decimal constant is the new value of the destination form pointer.

If the third format is used, the value of the numeric variable is the new value of the destination form pointer. If the value is fractional, it is truncated.

If the fourth or fifth format is used, the new value of the destination form pointer is obtained by the formula:

(*decimal value of form-pointed character or literal character of second operand*) - 31

If the sixth format is used, the expression is evaluated and the result determines the value to which the form pointer will be reset. For a character expression, the result is considered to be a literal character, and the new form pointer value is obtained by the preceding formula. For a numeric expression, the numeric result is the value to which the destination form pointer will be reset. If the value is fractional, it is truncated.

If the new form pointer is less than zero, then no change occurs and the eos flag is set. If the new form pointer is greater than the maximum length of the destination variable, then the form pointer is set to the maximum length of the destination variable and the eos flag is set. If the form pointer is set to a value greater than the logical length pointer of the destination variable, then the length pointer is set to the same value as the form pointer and the eos flag is set.

resetparm

label **`resetparm`**

Flags affected: none

See also: call, cverb, routine, getparm, verb

The resetparm statement resets the getparm position so that the next getparm statement executed will retrieve the first optional parameter and value.

retcount

label **`retcount`** *numvar*

numvar is the destination variable

Flags affected: equal, over

The retcount statement moves the number of entries in the return stack into the destination variable. If that number is zero, then the equal flag is set. Otherwise, the equal flag is cleared. If the number is truncated when it is moved into the destination, the over flag is set. Otherwise, the over flag is cleared.

return

label **`return`**
label **`return`** **`if`** *cond*
label **`return`** **`if not`** *cond*
label **`return`** **`if`** *function-key*
label **`return`** **`if not`** *function-key*
label **`return`** **`if`** *expression*
label **`return`** **`if not`** *expression*

cond is one of equal, less, over, eos or greater
function-key is one of **`F1`**, **`F2`**, **`F3`**, **`F4`**, **`F5`**, **`F6`**, **`F7`**, **`F8`**, **`F9`**, **`F10`**, **`F11`**, **`F12`**, **`F13`**, **`F14`**, **`F15`**, **`F16`**, **`F17`**, **`F18`**, **`F19`**, **`F20`**, **`up`**, **`down`**, **`left`**, **`right`**, **`insert`**, **`delete`**, **`home`**, **`end`**, **`pgup`**, **`pgdn`**, **`tab`**, **`bktab`**, **`esc`**, or **`enter`**
expression is an algebraic expression

Flags affected: none

See also: call, noreturn, perform, setendkey

The return statement causes program execution to continue with the statement whose address is taken from the top of the return stack.

If the first format is used, the return is unconditional. If the return is unconditional, execution continues with the statement from the top of the return stack.

If one of the other formats is used, execution continues with the statement from the top of the return stack only if the condition tested by an **`if`** is true or the condition tested by an **`if not`** is false. If a condition is not met, program

execution continues with the statement that follows the return statement.

The *function-key* form of the return statement is only applicable following a keyin statement that was interrupted when a function key was pressed. Each *function-key* condition may only be tested once with a goto, call, or return operation. All *function-key* conditions are reset after they are checked and at the start of keyin execution.

When the return instruction transfers execution to a statement from the return stack, that entry is removed from the stack.

If a program attempts a return when the return stack is empty, an E 503 error will occur.

rollout, clientrollout

label **`rollout`** *charexp*
label **`clientrollout`** *charexp*

charexp is the source operand

Flags affected: over

The rollout statement causes the command line in the logical string of the source operand to be executed by the operating system command interpreter (or shell). A new task is created to execute the command line. DB/C program execution waits for the successful or unsuccessful completion of that task.

When running under Smart Client, the clientrollout statement causes the command line in the logical string of the source operand to be executed by the operating system command interpreter on the client computer. A new task is created to execute the command line. DB/C program execution waits for the successful or unsuccessful completion of that task. When not running under Smart Client, the clientrollout statement does nothing except that it causes the over flag to be set.

Before execution of the command line, all file and record locks are released and all non-exclusively opened files are closed at the operating system level. The files will remain logically opened in the DB/C program and will be reopened at the operating system level on the first I/O statement encountered.

If the source operand is a null string or if the rollout statement detects an execution error, over is set. Program execution then resumes at the next statement.

rotate

label **`rotate`** *numexp prep numvar*

numexp is the source operand
numvar is the destination operand

Flags affected: equal

The destination operand is converted to a 32 bit integer that is the value to be rotated. The source operand is converted to an integer, by truncation if necessary, that is the number of bits that the destination will be rotated. If the number of bits to rotate is positive, then the rotation is toward the left (toward the higher order bits). If the number of bits is negative then the rotation is to the right. Bits rotated out of the high order bit position are rotated into the low order bit position, and vice versa. The result is moved to the destination operand. If the result is zero, the equal flag is set. Otherwise, the equal flag is cleared.

routine, lroutine, endroutine

label **`routine`** *list*
label **`lroutine`** *list*
`endroutine`

list is a destination list of address variables

Flags affected: none

See also: call

The routine and lroutine statements define program labels that can be used as routine entry points. The two statements operate identically, except that the label of a routine statement is visible to other compiled modules while the label of an lroutine statement is only visible locally.

Execution of a routine or lroutine statement does nothing unless the statement is called by a call statement with parameters. In this case, the address of each variable in the calling list is moved into the corresponding address variable in the routine list.

If the calling list contains more variables than are in the routine list, then excess variables are ignored. If the routine list contains more variables, then excess address variables are left unchanged. If the routine statement is executed by any means other than a call statement with parameters, then the list of address variables is left unchanged.

A typeless (var) address variable may be included in the destination list of a routine or lroutine statement.

A label address variable may be included in the destination list of a routine or lroutine statement if the variable name is prefixed with the tilde (**~**) character.

An endroutine statement ends the scope of variables declared in the routine. A variable defined within the scope of a routine/endroutine or an lroutine/endroutine is called a local variable. A local variable is not usable after its corresponding endroutine statement is encountered. A local variable is visible in routines that are nested within the routine/endroutine in which the variable is declared. Two different local variables can have the same name as long as they are both not visible at the same place in the program.

The definition of a local address variable that is used as a parameter in a routine or lroutine statement may immediately follow the routine or lroutine statement. This is an exception to the rule which states that a variable must be declared before it is used.

scan

label **`scan`** *charexp prep charvar*
label **`scan`** *hexcon prep charvar*

charexp is the source operand
hexcon is the source operand
charvar is the destination operand

Flags affected: equal, eos

The scan statement searches for a string of characters within a character variable.

The search string is determined by the source operand. If the source is character, the search string is the logical string. If the source is numeric, the search string is the character represented by that character code.

The logical string of the destination operand is searched for the first occurrence of the search string. The search begins with the form pointed character of the destination. If the search string is found in the destination, the equal flag is set, the form pointer of the destination variable is set to point to the first matched character, and the logical length pointer is left unchanged. If no match is found, the equal flag is cleared and the pointers remain unchanged.

If the form pointer of either operand is zero, the eos flag is set, and the form pointer is not changed.

If the length of the logical string of the source is greater than the length of the logical string of the destination, no match can occur.

scrnrestore

label **`scrnrestore`** *charvar*

`scrnrest` may be used in place of **`scrnrestore`**

charvar is the source operand

Flags affected: none

See also: scrnsave, scrnsize

The scrnrestore statement restores the screen image and video state information from a character variable. The variable must have been filled by a prior scrnsave statement. If the scrnrestore operation is unable to restore the screen image, an E 506 error occurs.

scrnsave

label `scrnsave` *charvar*

charvar is the destination operand

Flags affected: eos

See also: scrnrestore, scrnsize

The screen image and all screen state information are stored in the destination variable starting at the first character of the variable. Screen state information includes the number of lines on the screen, the screen size, the current subwindow settings, the current cursor position, the current cursor type (e.g., underline, block, or invisible), the echo secret character, the keyin cancel character, the keyin timeout value, the current display at tributes (e.g., ***revon**, ***ulon**, etc.), the current keyin attributes (e.g., ***editon**, ***it**, etc.), the current ending key list, and the screen image.

The destination form pointer is set to one and the logical length pointer is set to the number of characters moved. The destination operand must be large enough to contain the information being saved. The size required may be ascertained with the scrnsize statement. If the destination variable is not big enough to store all the information, the eos flag is set. Otherwise, the eos flag is cleared.

scrnsize

label `scrnsize` *numvar*

numvar is the destination operand

Flags affected: none

See also: scrnrestore, scrnsave

The scrnsize statement moves the number of characters needed to store the screen image and all screen state information into the destination variable. This number is the minimum size required for the destination variable in the scrnsave statement.

Screen state information includes the number of lines on the screen, the screen size, the current subwindow settings, the current cursor position, the current cursor type (e.g., underline, block, or invisible), the echo secret character, the keyin cancel character, the keyin timeout value, the current display attributes (e.g., ***revon**, ***ulon**, etc.), the current keyin attributes (e.g., ***editon**, ***it**, etc.), and the screen image.

search

label **`search`** *exp prep var prep numexp prep numvar*

exp is the key
var is the first variable in a list of contiguous variables
numexp is the number of variables in the list of contiguous variables
numvar is the destination variable

Flags affected: equal, over

The search statement compares a key variable to a list of contiguous data variables. If the search is successful, a number is moved to the destination variable which represents the relative position of the variable in the list that matched the key.

Each variable from the list of contiguous variables is compared to the key. The second operand, *var*, specifies the first variable in the list to be searched. The number of variables to search is the value of *numexp*. If the number is fractional, it is truncated (not rounded).

The comparison differs from the one performed in the match operation. The search begins with the variable specified in the second operand and continues through contiguous variables until the search succeeds or until the number of items to be searched is exhausted.

If the search is successful, the equal flag is set and the over flag is cleared. Furthermore, the relative position in the list of the item matching the key is moved to the destination numeric variable. For instance, if a match is found in the fifth item of the list, a five is moved to the destination.

If the search fails, the equal flag is cleared, the over flag is set, and the destination variable is set to zero.

If the key is a character variable, its logical string is the search string. The search string is compared to the logical string of a variable from the list. If the length of the logical string of the key is greater than the length of the logical string of the list variable, the search fails. If the logical length of the key is less than or equal to the length of the variable and all characters in the key are found in the variable, the search succeeds. If any character in the sequence does not match, the search fails.

If the key is a numeric variable or expression and the variable from the list is a numeric variable, then the numeric value of the key is compared to the numeric value of a variable from the list. If the values are the same, the search succeeds. If the values are not the same, the search fails.

If the key is numeric and the variable from the list is a character variable, then the matching process works in the same fashion as if the key is a character variable.

send

label `send` *cfile* `,` *numvar* `;` *list*

cfile is the label of a comfile declaration
numvar is the timeout value
list is a list of character and numeric variables, literals and control codes

Flags affected: none

See also: comclr, comtst, recv, sendclr

The send statement initiates the transmission of data. *list* is the list of variables, literals, and control codes that make up the message that is transmitted. When the send is initiated, the send status of the comfile is changed to "send pending".

The timeout value is the number of seconds the send will remain "send-pending" before it is canceled. If the timeout value is -1, then the send will never timeout. If the timeout value is zero, the send will timeout immediately if the send cannot be completed immediately.

The send status of a comfile is set to "send-clear" before an attempt to transmit data is made. Therefore, if the status of a comfile is "send-pending" and another send for the same comfile is executed, the first send will be canceled.

When the send has completed, the status of the sending comfile is changed to "send complete". If the send does not complete successfully, then a timeout occurs and the status is changed to "send-timeout". The status of a comfile can be checked using the comtst statement.

***sl** is the blank suppression control code. This control code affects all remaining character variables in the list. This control code causes no blanks to be sent for the characters between the length pointer and the maximum length of the string. If the variable is cleared, then nothing is sent.

***ll** is the logical length control code. This control code is in effect for all character variables remaining in the list or until a ***pl** control code is encountered. The logical string of the character variable is sent.

***pl** is the physical length control code. This control code cancels the effect of the ***ll** and ***sl** control codes.

sendclr

label **`sendclr`** *cfile*

cfile is the label of a comfile declaration

Flags affected: less

See also: send

The sendclr statement sets the send status of the comfile to "send-clear". It does not affect the receive status. If the status prior to execution of the sendclr statement was "send-pending", then the send is canceled and the less flag is set.

set

label **`set`** *list*

list is a comma delimited list of character and numeric variables, arrays, and list variables

Flags affected: none

See also: clear

The set statement sets the value of each element of the list to "1". In the case of a numeric variable, the value is set to 1. In the case of a character variable, the form pointer and length pointer are set to one and the first position of the variable is set the character "1". If an entry is a list variable, then all elements of the list are set to "1". If an entry is an array, then all variables of the array are set to "1". All other types of variables are ignored.

setendkey

label **`setendkey`** *list*

list is a comma delimited list of numeric variables, numeric arrays, decimal constants, and equate labels

Flags affected: none

See also: getendkey, clearendkey, keyin

The setendkey statement enables the keys specified by the numeric values in the list as keyin ending keys. The ending key values are described in the Keyboard and Display Manipulation Statements section of the DB/C Programming Language General Information chapter.

Zero and negative values in the list are ignored.

The current keyin ending key status is saved and restored by the statesave, staterestore, scrnsave, and scrnrestore statements.

setflag

label **`setflag`** *cond*
label *setflag* **`not`** *cond*

cond is one of eos, equal, less, over

Flags affected: eos, equal, less, or over

The setflag statement alters the value of a flag. If the first format is used, the specified flag is set. If the second format is used, the specified flag is cleared.

setlptr

label **`setlptr`** *charvar1*
label **`setlptr`** *charvar1 prep dcon*
label **`setlptr`** *charvar1 prep numvar*
label **`setlptr`** *charvar1 prep charvar2*
label **`setlptr`** *charvar1 prep charlit*
label **`setlptr`** *charvar1 prep expression*

charvar1 is the destination operand
dcon is the new length pointer value
numvar contains the new length pointer value
charvar2 is the character variable used to determine the new length pointer value
charlit is the character literal used to determine the new length pointer value
expression is an expression used to determine the new length pointer value

Flags affected: over, eos

The setlptr statement sets the length pointer of the destination character variable.

If the first format is used, the new length pointer value is the maximum length of the destination variable.

If the second format is used, the decimal constant is the new length pointer value.

If the third format is used, the contents of the numeric variable are truncated (not rounded) to obtain the new length pointer value.

If the fourth or fifth format is used, the new value of the destination length pointer is obtained by the formula:

(*decimal value of length-pointed character or literal character of second operand*) - 31

If the sixth format is used, the expression is evaluated and the result determines the new value of the length pointer. For a character expression, the result is considered to be a literal character, and the new length pointer value is obtained by the preceding formula. For a numeric expression, the numeric result is the value to which the destination length pointer will be set. If the value is fractional, it is truncated.

If the value of the new length pointer is less than the value of the form pointer, then both pointers are set to the new value and the eos flag is set. The over flag is set if the new logical length pointer value would be less than zero or greater than the maximum length of the string. When the over flag is set, no change is made to the form pointer or length pointer of the destination.

setnull

label **`setnull`** *list*

list is a comma delimited list of character and numeric variables, arrays, and list variables

Flags affected: none

The setnull statement sets the value of each element of the list to the NULL value. Variables other than character and numeric variables are ignored.

sformat

label **`sformat`** *charvar prep numvar*
label **`sformat`** *charvar prep dcon*
label **`sformat`** *chrarray prep numvar*
label **`sformat`** *chrarray prep dcon*

charvar is the destination operand
chrarray is the destination operand
numvar contains the value of the new maximum length
dcon is the value of the new maximum length

Flags affected: over

The sformat statement changes the physical length of a character variable or of each element in a character variable array. The maximum length of the destination character variable is changed to the new maximum length value specified by the second operand.

If the first or third format is used, the new maximum length value is obtained from the numeric variable by truncation, not rounding. The form pointer and the logical length pointer of the destination are set to zero.

The maximum length can never be changed to a value that is larger than the declared maximum length that was specified during compilation. If this is attempted, the over flag is set and no change is made to the maximum length.

The form pointer and the length pointer of the destination operand are set to zero.

show

label	**`show`**	*resource prep device*
label	**`show`**	*image prep device*
label	**`show`**	*resource prep device prep numexp1 prep numexp2*
label	**`show`**	*image prep device prep numexp1 prep numexp2*

resource is the source operand
image is the source operand
device is the destination operand
numexp1 is the horizontal position
numexp2 is the vertical position

Flags affected: none

See also: hide

The resource or image specified as the source operand is made visible on the destination device variable. If the device is a window, and if images and panel resources occupy the same area of a window, images are shown behind the panels. The layering of images that overlap with each other is undefined, and the layering of panel resources that overlap with each other is undefined.

The third format allows a resource to be displayed at a particular location. *numexp1* and *numexp2* represent the horizontal and vertical positions of the upper left corner of the resource. For panel and popupmenu resources, this position is relative to the device. For menu and toolbar resources, the position operands are ignored. For dialog resources, the positions are relative to the screen. Also for dialog resources, if either position value is negative then they are both ignored and the dialog will be centered on the screen. Negative position values are meaningful for panel and popupmenu resources. For a panel resource, whether the positions are negative or not, only the part of the panel that appears within the boundaries of the window device will be visible. That is, it will be cropped at the edges of the parent window. A popupmenu will be completely visible even if part or all of it is outside of the window device.

The fourth format allows an image to be displayed at a particular location. *numexp1* and *numexp2* represent the horizontal and vertical coordinates of the upper left corner of the image. The position is relative to the current window. Images must be displayed within the current window. In the non-graphical versions of DB/C DX, this format of the show statement causes a runtime error to occur.

shutdown

label **`shutdown`**
label **`shutdown`** *charexp*

charexp is the source operand

Flags affected: none

The shutdown statement terminates program execution, closes all files, and returns control to the operating system. If the source operand is specified, it is ignored.

sound

label `sound` *numexp1 prep numexp2*

numexp1 is the first operand
numexp2 is the second operand

Flags affected: none

See also: beep

The first operand determines the pitch of the sound and the second operand determines the duration of the sound in tenths of a second. As the value of the first operand decreases, the tone gets higher.

The sound statement may behave like the beep statement on some terminals.

splclose

label `splclose`
label `splclose` *pfile*
label `splclose` *charexp*
label `splclose` *pfile*`,` *charexp*

pfile is the label of a pfile declaration
charexp is the option operand

Flags affected: none

The splclose statement causes printed output for the current spool file or device to revert to the default printer. If the splclose is for a device, the device is deallocated.

If the second or fourth form is used, the splclose statement logically disconnects the pfile variable from the device or file.

If the third or fourth form is used, the option operand contains a list of spool parameters separated by commas. Certain options may be operating system dependent. The following are valid spool parameters:

D is the delete file option. In conjunction with the **SUBMIT** option, this option causes the spool file to be deleted after the file has been printed.

SUBMIT or **SUBMIT=***string* This option sends the spool file to be printed. The optional *string* is the name of the printer to which the spooled file is sent.

splopen

label `splopen` *charexp1*
label `splopen` *pfile* `,` *charexp1*
label `splopen` *charexp1* `,` *charexp2*
label `splopen` *pfile* `,` *charexp1, charexp2*

charexp1 is the spool file name or device
charexp2 is the option operand
pfile is the label of a pfile declaration

Flags affected: over

See also: print, splclose

The splopen statement redirects the data that is printed by print statements to a different print file or device. The *charexp1* operand contains the name of the file or device to which print output will be directed. The *charexp2* operand contains one or more spool options.

The splopen operation for a device does not allocate the device. The first print statement allocates the device. If the device is already allocated to another user, then the print statement will wait until the printer has been deallocated. The print device is deallocated by means of the release statement.

If the splopen is for a file, then the one or more records are written for each print line. The file is a runtime operating system type text file. Standard carriage control characters are placed in the first character of each record. The carriage control characters are: **1** (top of form), **+** (no vertical skip) and blank (next line).

If the splopen is for a device, then device control characters are sent to the printer. The device control characters are: LF (line feed only), CR (carriage return only), and FF (form feed).

If the first or second form of the statement is used and no extension is given in *charexp1*, then the extension **.prt** is assumed.

If the second or fourth form of the statement is used, then the splopen statement logically connects the pfile variable to the device or resource specified by *charexp1*.

If the third or fourth form of the statement is used, then the characters in the logical string of the option operand are used to modify the operation of the splopen and following print statements. Multiple options are separated with commas. Certain options may be operating system dependent.

The following paragraphs describe the valid option characters.

A is the allocate device option. If the splopen is for a device, then the allocation of that device takes place immediately if it is not allocated to another user. If the device is allocated, then the over flag is cleared. If the device is already allocated to another user, the splopen fails and the over flag is set. The behavior of allocating a device may be operating system dependent. If the splopen is for a file, then the file is created only if it did not previously exist. If it did exist, splopen fails and the over flag is set. If it did not exist, the file is created and the over flag is cleared. If this option is not used, the over flag is unchanged.

B=*bin* is the paper source option. *bin* may contain spaces. The end of *bin* is indicated by a comma or the end of the option string. A comma may be embedded in the bin name by preceding it with a backslash. Acceptable values of bin are different for each environment. Use the getpaperbins statement to determine what values of bin will work for a given print destination.

BANNER is the banner option. This option causes a banner page to be printed before the print job. This option is only available for Linux when running without CUPS.

BOTTOMMARGIN=*n* is the bottom margin (in inches) for PDF and PS print output. The default is .25.

DEST=SERVER or **DEST=CLIENT** are the printer destination options for Smart Client. **DEST=SERVER** causes print data to be directed to a print file or device on the server. **DEST=CLIENT** causes print data to be directed to a print file or device local to the client. This option is only checked for if running under Smart Client. It will override the **dbcdx.print.destination** setting in DB/C DX config file.

C or **H** is the compressed file output option. If the splopen is for a file, then the file created is a compressed DB/C type file, instead of a runtime operating system text file. This option is mutually exclusive with the **D** option.

D or **R** is the device control characters option. If the splopen is for a file, then device control characters are placed in the print file instead of carriage control characters. This option is mutually exclusive with the **C** option.

I is the ignore form feed option. This option suppresses trailing form feeds.

J is the job setup option.

L=PDF, **L=PS**, **L=PCL**, **L=PCL(NORESET)**, **L=NATIVE**, and **L=NONE**, are the print output language op tions. **L=PDF** causes the output to be in Adobe PDF format. **L=PS** causes the output to be in PostScript format. **L=PCL** and **L=PCL(NORESET)** cause the output to be in PCL format. The **NORESET** option causes there to be no initialization data in the output. **L=NATIVE** causes the output to be done using the native operating system interface. **L=NONE** causes output to be basic line oriented output. **L=NONE** is typically used in conjunction with the **D** option and may be required when directly printing printer escape sequences. See the sections on Windows and Linux Considerations for operating system specific details.

LEFTMARGIN=*n* is the left margin (in inches) for PDF and PS print output. The default is .25.

MARGIN=*n* is the margin (in inches) for PDF and PS print output for all 4 sides. The default is .25.

N=*n* is the number of copies option. *n* is the number of copies to be printed.

O=LANDSCAPE or **O=PORTRAIT** are the paper orientation options.

P is the pipe option. This option is applicable only to Linux systems.

Q is the append option. If the splopen is for a file, then this option causes the print output to be appended to the end of the file if it previously existed. The default operation is to overwrite an existing file.

RIGHTMARGIN=*n* is the right margin (in inches) for PDF and PS print output. The default is .25.

S=*size* is the paper size option. size is the name of the paper size.

TOPMARGIN=*n* is the top margin (in inches) for PDF and PS print output. The default is .25.

W is the two-sided print option.

X is the no extension option. This option is applicable only if the output is to a file. If this option is included, then no extension is added to the first operand. Otherwise, if the first operand does not have an extension, **.prt** will be added.

Y=*text* is the document name option. If not specified it defaults to **DB/C**.

Z=*decimal-number* This option sets the dots-per-inch for PDF, PCL, and PostScript output formats. If not specified it defaults to 300. For PCL output the decimal number is one of: 96, 100, 120, 144, 150, 160, 180, 200 225, 240, 288, 300, 360, 400, 450, 480, 600, 720, 800, 900, 1200, 1440, 1800, 2400, 3600, 7200.

splopt

label `splopt` *charexp*

charexp is the option operand

Flags affected: over

The splopt statement specifies spool file options.

S is the printer setup option. In a GUI environment, this option causes the printer setup dialog box to be displayed. If the user cancels the printer setup dialog box, the over flag is set to true. Otherwise, it is set to false. This option is ignored in a non-GUI environment.

sqlcode

label **`sqlcode`** *numvar*

numvar is the destination operand

Flags affected: none

The SQL return code from the most recently executed SQLEXEC statement is placed to the destination operand.

sqlexec

label **`sqlexec`** *charexp*
label **`sqlexec`** *charexp from from-list*
label **`sqlexec`** *charexp into into-list*
label **`sqlexec`** *charexp from from-list into into-list*

charexp is the source operand
from-list is a list of source literals and variables
into-list is a list of destination variables

Flags affected: less, over, equal

The sqlexec statement allows any SQL statement to be executed. The source operand contains the SQL statement.

If the second or fourth format is used, then the *from-list* contains DB/C variables that replace variables in the SQL statement that are of the form **:**_n_ or **:**_nn_.

If the third or fourth format is used, then the *into-list* is used to return variables that were fetched by the SQL statement.

If the SQL return code field is negative, then the less flag is set and the over and equal flags are cleared. If the SQL return code is zero, then the equal flag is set and the less and over flags are cleared. If the SQL return code is the positive number 100, then the over flag is set and the less and equal flags are cleared. Note: 100 is the return code for row not found or no more rows to fetch.

sqlmsg

label **`sqlmsg`** *charvar*

charvar is the destination variable

Flags affected: none

The SQL error message from the most recently executed SQLEXEC statement is placed to the destination operand.

squeeze

label **`squeeze`** *charexp prep charvar*

charexp is the source operand
charvar is the destination operand

Flags affected: eos

The squeeze statement moves all non-blank characters in the logical string of the source operand to the destination operand. The destination form pointer is set to one, and the logical length pointer is set to the number of characters moved. If the destination is not large enough to contain all non-blank characters from the source, then as many characters as will fit are moved, and the eos flag is set.

stateresotre

label **`staterestore`** *charvar*

`staterest` may be used in place of **`staterestore`**

charvar is the source operand

Flags affected: none

See also: statesave, statesize

The screen state information stored in the source operand is restored. The variable must have been filled by a prior statesave statement. If the staterestore operation is unable to restore the screen state, an E 506 error occurs.

statesave

label `statesave` *charvar*

charvar is the destination operand

Flags affected: eos

See also: staterestore, statesize

The screen state information is stored in the destination variable starting at the first character of the variable. The screen state information includes the number of lines on the screen, the screen size, the current subwindow settings, the current cursor position, the current cursor type (e.g., under line, block, or invisible), the echo secret character, the keyin cancel character, the keyin timeout value, the current display attributes (e.g., ***revon**, ***ulon**, etc.), the current keyin attributes (e.g., ***editon**, ***it**, etc.), and the current ending key list.

The destination form pointer is set to one and the logical length pointer is set to the number of characters moved. If the destination variable is not big enough to store all the information, the eos flag is set. Otherwise, the eos flag is cleared.

statesize

label **`statesize`** *numvar*

numvar is the destination operand

Flags affected: none

See also: statesave

The statesize statement moves the number of characters needed to store the screen state information into the destination. This number is used to deter mine the size of the destination variable in the statesave instruction.

The screen state information includes the number of lines on the screen, the screen size, the current subwindow settings, the current cursor position, the current cursor type (e.g., underline, block, or invisible), the echo secret character, the keyin cancel character, the keyin timeout value, the current display attributes (e.g., ***revon**, ***ulon**, etc.), the current keyin attributes (e.g., ***editon**, ***it**, etc.) and the screen image.

stop

label **`stop`**
label **`stop`** **`if`** *cond*
label **`stop`** **`if not`** *cond*
label **`stop`** **`if`** *function-key*
label **`stop`** **`if not`** *function-key*
label **`stop`** **`if`** *expression*
label **`stop`** **`if not`** *expression*

cond is one of equal, less, over, eos or greater
function-key is one of **`F1`**, **`F2`**, **`F3`**, **`F4`**, **`F5`**, **`F6`**, **`F7`**, **`F8`**, **`F9`**, **`F10`**, **`F11`**, **`F12`**, **`F13`**, **`F14`**, **`F15`**, **`F16`**, **`F17`**, **`F18`**, **`F19`**, **`F20`**, **`up`**, **`down`**, **`left`**, **`right`**, **`insert`**, **`delete`**, **`home`**, **`end`**, **`pgup`**, **`pgdn`**, **`tab`**, **`bktab`**, **`esc`**, or **`enter`**
expression is an algebraic expression

Flags affected: none

See also: setendkey

The stop statement terminates program execution, closes all files, and returns control to the operating system. Operation of the stop statement may be altered by the dbcdx.stop runtime property.

The stop statement in the first format is unconditional.

If one of the other formats is used, the stop statement is executed only if the condition tested by an **if** is true or the condition tested by an **if not** is false. If the condition is not met, program execution continues with the next statement.

The *function-key* form of the stop statement is only applicable following a keyin statement that was interrupted when a function key was pressed. Each function-key condition may only be tested once with a stop statement. All *function-key* conditions are reset after they are checked and at the start of keyin execution.

store

label **`store`** *exp prep numexp prep list*
label **`store`** *charexp prep device*
label **`store`** *image prep device*

exp is the source variable, literal, or exp ression
numexp is the index
list is a list of character variables, numeric variables, arrays or list variables
lstvar defines the list of variables
charexp is the source operand
image is the source operand
device is the destination operand

Flags affected: equal, less, over, eos (both operands numeric); eos (one or both operands character); none (image and device variables are operands)

See also: load, move

In the first form, the store statement moves the source variable or literal into the indexed position of the list. The value of the index (*numexp*) determines which member of the destination list will store the source value. Each element of an array or list variable is counted as one variable.

The destination is the Nth variable in the list where the index defines N. The index is truncated to an integer value, not rounded. If the value N is less than one or greater than the number of variables in the list, then the store statement does nothing.

After a variable is selected as the destination, an implicit move operation is performed on the source. Flags are set according to the rules used with the move instruction.

In the second form, the store statement copies the text that is the logical string of the source operand to the destination device.

In the third form, the store statement moves an image from the image variable to the destination device. In non-graphical versions of DB/C DX, this form of the store statement causes a runtime error to occur.

storeadr

label **`storeadr`** *adrvar prep numexp prep list*
label **`storeadr`** *adrvar prep numexp prep lstvar*
label **`storeadr`** *adrvar prep numexp prep array*

adrvar is the source operand
numexp is the index
list is a list of destination address variables.
lstvar is a list variable
array is an array of address variables

Flags Affected: none

See Also: loadadr, moveadr

The source operand may be any type of variable except a label variable. The index (*numexp*), used to specify the destination, is truncated to an integer value, not rounded. If the index is less than one or greater than the number of variables in the list, then the storeadr statement does nothing.

If the first format is used, the storeadr statement moves the address of the source variable into a specified address variable from the list. The destination address variable within the list must be the **@** form of the source operand or a typeless address variable. The value of the index (*numexp*) deter mines which variable in the list is the destination. A list variable or an array of address variables is not valid in the destination list. A list **@** address variable is valid within the destination list.

If the second or third format is used, the storeadr statement moves the address of the source operand into a specified address variable within the given list or array of addresses (*lstvar* or *array*). The value of the index (*numexp*) determines which variable within the given list or which element within the given array is the destination. The destination, within the list or array, must be the **@** form of the source variable or a var **@** variable.

storelabel

label **`storelabel`** *lblvar prep numexp prep list*
label **`storelabel`** *prog-label prep numexp prep list*

lblvar is the source label variable
numexp is the index
prog-label is the source program label
list is a list of label variables

Flags affected: none

See also: loadlabel, movelabel

The storelabel statement stores the address of the source in a label variable in the destination list. The value of the index determines which member of the destination list will store the address of the source label. The destination is the Nth variable from the list where the index defines N. The index is truncated to an integer value, not rounded. If N is less than one or greater than the number of variables in the list, then storelabel does nothing.

subtract

label **`subtract`** *numexp1 prep numvar*
label **`subtract`** *numexp1 prep numexp2 giving numvar*
label **`subtract`** *numexp1 prep numarray3*
label **`subtract`** *numarray1 prep numvar*
label **`subtract`** *numarray1 prep numarray3*
label **`subtract`** *numarray1 prep numarray2 giving numarray3*

`sub` may be used in place of **`subtract`**

numexp1 is the first source operand
numexp2 is the second source operand
numvar is the destination operand
numarray1 is the first source operand
numarray2 is the second source operand
numarray3 is the destination operand

Flags affected: equal, less, over

If the first format is used, the source operand is subtracted from the destination operand and the result is placed in the destination operand.

If the second format is used, then the first source operand is subtracted from the second source operand and the result is placed in the destination operand.

If the third format is used, the source operand is subtracted from each element of the destination operand.

If the fourth format is used, each element of the source array is subtracted from the destination operand. The result is placed in the destination operand.

If the fifth format is used, each element of the source array is subtracted from the corresponding element in the destination array.

If the sixth format is used, each element of the first source array is subtracted from the corresponding element of the second source array, and the result is placed in the corresponding element of the destination array.

Rounding takes place when the intermediate result is moved to the destination.

switch, case, default, endswitch

label **`switch`** *var*
label **`case`** *exp*
label **`case`** *exp* **`or`** *exp ...*
label **`default`**
label **`endswitch`**

The or bar character (**`|`**) may be used in place of **`or`**. A case statement may be continued on the next line if a colon is used in place of the **`or`** or **`|`** character.

var is the switch variable
exp is the case comparison operand
... means that **`or`** *exp* may be repeated

Flags affected: none

The switch, case, default, and endswitch statements conditionally control execution of program lines between the switch and endswitch statements.

The switch statement is required and must precede the other statements. No statements are allowed between the switch statement and the first case statement. The case statement may be specified any number of times, but must be specified at least once. The optional default statement must follow the last case statement. The required endswitch statement follows all other statements within the scope of the switch statement.

Zero or more DB/C program statements may follow each case and default statement. These statements are executed if the case comparison operand matches the switch variable. If there are two or more case comparison operands separated by **or**, the statements following the case statement are executed if the switch variable matches any of the case comparison operands. If none of the case comparison operands matches the switch variable, then the statements following the default statement are executed.

Each switch/endswitch group of statements may be nested inside a case or default statement group up to 32 levels deep.

tabpage

label `tabpage`
Flags affected: none

See also: set, clear

The tabpage statement does nothing.

test

label **`test`** *var*

> *var* is the source variable. It may be a numeric variable, a character variable, a file variable, an ifile variable, an afile variable, a device variable, a resource variable, or an object variable.

Flags affected: equal, less (numeric source); eos (character source)

See also: set, clear

If the source variable is a numeric variable, the flags are set as follows: If the value is zero, the equal flag is set. If the value is non-zero, the equal flag is cleared. If the value is less than zero, the less flag is set. If the value is greater than or equal to zero, the less flag is cleared.

If the source variable is a character variable, then the flags are set as follows: If the form pointer is zero, the eos flag is set. If the form pointer is non-zero, the eos flag is cleared.

If the source variable is a file variable, an ifile variable, an afile variable, a device variable or a resource variable, the over flag is cleared if it is open. Otherwise, the over flag is set.

If the source variable is an object variable, the over flag is cleared if the object variable is instantiated. Otherwise, the over flag is set.

testadr

label `testadr` *adrvar*

adrvar is the address variable

Flags affected: over

The testadr statement tests the validity of an address variable. The over flag is set if the address contained is invalid. Otherwise, the over flag is cleared.

testlabel

label **`testlabel`** *lblvar*

lblvar is the label variable

Flags affected: over

The testlabel statement tests the validity of a program label variable. The over flag is set if the label is invalid. Otherwise, the over flag is cleared.

trap

label **`trap`** *prog-label* **`if`** *allkeys*
label **`trap`** *prog-label* **`if`** *allchars*
label **`trap`** *prog-label* **`if`** *allfkeys*
label **`trap`** *prog-label* **`if`** *event*
label **`trap`** *prog-label trap-option* **`if`** *event*
label **`trap`** *prog-label* **`giving`** *charvar trap-option* **`if`** *event*
label **`trap`** *prog-label trap-option* **`giving`** *charvar* **`if`** *event*

prog-label is a program execution label
trap-option is one or more of **`noreset`**, **`prior`**, **`nocase`**, or **`disable`**
charvar is the destination character variable
event is one of **`range`**, **`format`**, **`cfail`**, **`io`**, **`spool`**, **`parity`**, **`debug`**, **`prtofl`**, **`queue`** *queue*, **`anyqueue`**, **`timeout`** *numexp*, **`timestamp`** *charexp*, **`interrupt`**, **`int`**, **`F1`**, **`F2`**, **`F3`**, **`F4`**, **`F5`**, **`F6`**, **`F7`**, **`F8`**, **`F9`**, **`F10`**, **`F11`**, **`F12`**, **`F13`**, **`F14`**, **`F15`**, **`F16`**, **`F17`**, **`F18`**, **`F19`**, **`F20`**, **`up`**, **`down`**, **`left`**, **`right`**, **`insert`**, **`delete`**, **`home`**, **`end`**, **`pgup`**, **`pgdn`**, **`tab`**, **`bktab`**, **`esc`**, **`backspace`**, **`bkspc`**, **`cancel`**, **`enter`**, **`char`** *charvar*, a character variable, or a one-character literal

Flags affected: none

See also: call, trapclr

The trap statement transfers control to the statement designated by *prog-label* when a specified event takes place. The *prog-label* is the program execution label where control will be transferred when the event occurs.

The *event* specifies what condition must occur for the trap to occur. For the forms of the trap instruction in which the *event* keyword does not appear, a key trap is set.

The *trap-options* are optional and alter the operation of the trap instruction.

The actual transfer of control to the trap routine does not happen until the event occurs or until a particular key is pressed. Execution of the trap statement only sets the trap.

After a trap occurs, the transfer of control is handled in a manner analogous to a call instruction. That is, the current execution point in the program is saved in the return stack. When a return statement is executed, the program resumes execution at the instruction following the statement that was executing when the trap occurred unless the trap occurs while reading from an afile. In this case, the program resumes execution at the beginning of the read statement.

If the first form of the trap instruction is used, the trap will occur when any key with the following value is trapped: decimal value 1-255, **F1-F20**, **Enter**, **Esc**, **Backspace**, **Tab**, **Back Tab**, **Up**, **Down**, **Left**, **Right**, **Insert**, **Delete**, **Home**, **End**, **Page Up**, and **Page Down**.

If the second form of the trap instruction is used, the trap will occur when any key with the following value is trapped: decimal value 32-255.

If the third form of the trap instruction is used, the trap will occur when any of the following keys is pressed: **F1-F20**, **Enter**, **Esc**, **Backspace**, **Tab**, **Back Tab**, **Up**, **Down**, **Left**, **Right**, **Insert**, **Delete**, **Home**, **End**, **Page Up**, and **Page Down**.

The last four forms of the trap instruction deal with particular events.

If the event is one of **range**, **format**, **cfail**, **io**, or **spool**, then the trap occurs when an execution error of one of these classes occurs.

If the event is **anyqueue**, then the trap occurs whenever any queue variable contains entries. The trap event queue occurs whenever the given queue variable contains an entry. The queue event and the noreset trap option are mutually exclusive. If both are specified, noreset is ignored.

The parity, debug, and prtofl events cannot occur.

With the timeout event, *numexp* specifies the number of seconds that must elapse before the trap occurs. The timeout resolution is one hundredth of a second. The timeout event is mutually exclusive with the noreset trap option. Only one timeout event may be concurrently active.

With the timestamp event, *charexp* specifies the date and time when the trap occurs. The format of *charexp* is the same as with clock timestamp. If *charexp* is shorter than 14 characters, it is logically extended with zeros. timestamp resolution is one hundredth of a second, unless modified by a configuration option. If timestamp specifies a time or date that has already passed, the trap will occur immediately. The timestamp event is mutually exclusive with the noreset trap option. Only one timeout event may be concurrently active.

If the event is char *charvar*, or simply a character variable, then the character pointed to by the form pointer is the trap character. If the event is a one-character literal, the character specified is the trap character. The event occurs when the corresponding keystroke is entered on the keyboard.

The other events are individual key events. These occur when the specified key stroke is entered on the keyboard.

The last three forms of the trap instruction include trap options.

The transfer of control occurs differently if the prior trap option is specified and the statement executing was keyin, wait, or comwait. In this case, the transfer of control is done as if the call had taken place before the statement that was executing when the event occurred.

When the trap action occurs, the trap for that event is cleared. However, if the noreset trap option is specified, the trap is not cleared. The queue event and the noreset trap option are mutually exclusive.

When the disable trap option is specified, the statements executing in the trap routine cannot be interrupted by keystroke events. The effect of the disable trap option ceases when an enable or keyin statement executes or program chaining occurs.

The trap for characters is case sensitive unless the nocase trap option is specified.

The giving *charvar* form of the trap statement causes information regarding the event to be stored in the destination character variable. For error events, the error message is stored. For character traps, the character entered at the keyboard is stored. For key traps, the name of the key is stored.

trapclr

label **`trapclr`** *event*
label **`trapclr`** *all*
label **`trapclr`** *allchars*
label **`trapclr`** *allfkeys*
label **`trapclr`** *allkeys*

event is one of **`range`**, **`format`**, **`cfail`**, **`io`**, **`spool`**, **`parity`**, **`debug`**, **`prtofl`**, **`queue`** *queue*, **`anyqueue`**, **`timeout`** *numexp*, **`timestamp`** *charexp*, **`interrupt`**, **`int`**, **`F1`**, **`F2`**, **`F3`**, **`F4`**, **`F5`**, **`F6`**, **`F7`**, **`F8`**, **`F9`**, **`F10`**, **`F11`**, **`F12`**, **`F13`**, **`F14`**, **`F15`**, **`F16`**, **`F17`**, **`F18`**, **`F19`**, **`F20`**, **`up`**, **`down`**, **`left`**, **`right`**, **`insert`**, **`delete`**, **`home`**, **`end`**, **`pgup`**, **`pgdn`**, **`tab`**, **`bktab`**, **`esc`**, **`backspace`**, **`bkspc`**, **`cancel`**, **`enter`**, **`char`** *charvar*, a character variable, or a one-character literal

Flags affected: none

See also: trap

If the first format is used, the trapclr statement clears the trap that was set by the trap statement for the designated event.

If the second format is used, then all traps are cleared.

If the third format is used, then all character traps are cleared.

If the fourth format is used, then all function key traps are cleared.

If the fifth format is used, then all character and function key traps are cleared.

traprestore

label **`traprestore`** *charvar*

`traprest` may be used in place of **`traprestore`**

charvar is the source operand

Flags affected: none

See also: trap, trapclr, trapsave, trapsize

The traprestore statement restores the trap settings previously saved in the source operand by the trapsave statement. Any traps that are currently set when the traprestore statement is executed are no longer valid (an implicit trapclr statement all is executed). Execution of a chain or stop statement causes any trap information stored previously by a trapsave statement to become invalid. If the traprestore statement is unable to restore the traps, an E 506 error occurs.

trapsave

label `trapsave` *charvar*

charvar is the destination operand

Flags affected: eos

See also: trap, traprestore, trapsize

The trapsave statement saves the states of all active traps in the destination. The current traps are not changed.

The destination must be large enough to contain the trap information. The trapsize statement can be used to determine the size of the destination variable. If the destination variable is not large enough to contain the trap information, the eos flag is set.

trapsize

label **`trapsize`** *numvar*

numvar is the destination operand

Flags affected: none

See also: trapsave

The trapsize statement moves the number of characters needed to store the current trap state information in the destination variable of a trapsave statement.

trim

label **`trim`** *charexp prep charvar*

charexp is the source operand
charvar is the destination operand

Flags affected: eos

All characters in the logical string of the source operand are moved to the destination except for leading and trailing blanks. The destination form pointer is set to one and the logical length pointer is equal to the number of characters moved. If the destination is not large enough to contain all characters to be moved, then as many characters that will fit are moved, and the eos flag is set.

type

label **`type`** *charvar*
label **`type`** *var*`,` *numvar1*
label **`type`** *var*`,` *numvar1*`,` *numvar2*
label **`type`** *var*`,` *numvar1*`,` *numvar2*`,` *numvar3*
label **`type`** *var*`,` *numvar1*`,` *numvar2*`,` *numvar3*`,` *numvar4*

charvar is the source operand
var is the source operand
numvar1 is the first destination operand
numvar2 is the second destination operand
numvar3 is the third destination operand
numvar4 is the fourth destination operand

Flags affected: equal, eos (only one operand); over (two or more operands)

If the first form of the type statement is used, the logical string of the source is checked to see if it is a valid DB/C number. If the logical string conforms to the rules defining valid numeric data, the equal flag is set. Otherwise, the equal flag is cleared. If the form pointer of the source is zero, the equal flag is cleared and the eos flag is set.

If any of the other forms of the type statement is used, the type of the source variable is checked and a corresponding number is moved to the first destination operand. If the source is a var (typeless) variable, then the variable to which it points is used. Values that may be moved into the first variable are:

0 if the *var* variable contains an invalid address
1 for a char (dim) variable
2 for a number (form), integer, or float variable
3 for a char (dim) array variable
4 for a number (form), integer, or float array variable
5 for a list or varlist variable
6 for a file variable
7 for an ifile variable
8 for an afile variable
9 for a comfile variable
10 for an image variable
11 for a device variable
12 for a resource variable
13 for a queue variable
14 for a label variable
15 for a pfile variable
16 for an object variable
17 for an array of typeless pointers

If the source is an array, the number of elements in each dimension is moved to the second, third, and fourth destination operands.

Any unused destination operands are set to zero.

If the value moved to the first destination operand is zero, then the over flag is set.

unlink

label **`unlink`** *resource*
label **`unlink`** *device*

resource is the source operand
device is the source operand

Flags affected: none

See also: link

The link between the specified resource or device and the queue to which it is linked is destroyed.

unload

label **`unload`**
label **`unload`** *charexp*

charexp is the source operand

Flags affected: none

See also: loadmod

The first format of the unload statement removes all instances of secondary modules that were not preloaded. If the unload statement is executed from a secondary module, then all instances and modules are unloaded except the current instance of the module containing the unload statement.

The second format of the unload statement unloads one or all instances of a secondary module. If the logical string of the source operand contains **< >** or **<***name***>**, then the unnamed or named instance of a module is unloaded. If **< >** and **<***name***>** are not specified, then all instances of the given module are unloaded. If the unload statement is being executed from one of the module instances to be unloaded, then all instances are unloaded except the current instance. The unload of preloaded module name is ignored.

unlock

label **`unlock`** *file*
label **`unlock`** *file* `,` *numexp*
label **`unlock`** *ifile*
label **`unlock`** *ifile* `,` *numexp*
label **`unlock`** *afile*
label **`unlock`** *afile* `,` *numexp*

file is the label of a file declaration
ifile is the label of an ifile declaration
afile is the label of an afile declaration
numexp is the source operand

Flags affected: none

The unlock statement unlocks one or all locked records in the file associated with the file, ifile, or afile variable.

If the form of the unlock statement with *numexp* is specified, the value is the value returned by fposit of the record to be unlocked. If the other form of the unlock statement is specified, all locked records are unlocked.

unpack

label **`unpack`** *exp prep list*

exp is the source operand
list is a list of character and numeric variables, arrays, and list variables

Flags affected: eos, over

The unpack statement distributes the contents of a variable among several variables. The logical string of the source operand is moved into the variables in the list.

Characters are moved from the logical string of the source operand to the first variable in the destination list until the maximum length of that variable has been reached. If the first variable in the list is a character variable, then the form pointer is set to one and the length pointer is set to the last character moved. If the first variable is a numeric variable, then the characters moved must exactly match the destination numeric format.

Characters from the source are then moved to the second variable in the same manner. This process continues until the variable list is exhausted or until the characters in the source operand are exhausted. If the characters in the source operand are exhausted before all variables of the list have been used, the form pointers of the remaining character variables are set to zero, and the values of the remaining numeric variables are set to zero.

If a destination variable is numeric and the characters from the source do not fit the numeric format exactly, then the over flag is set, and the value of the destination numeric variable is set to zero. Otherwise, the over flag is cleared.

If the form pointer of the source operand is zero, the eos flag is set, the form pointer of each character variable in the list is set to zero, and the value of every numeric variable in the list is set to zero.

unpacklist

label **`unpacklist`** *lstvar prep list*

lstvar is the source list variable
list is the destination list of address vari ables and address variable arrays

Flags affected: none

The unpacklist statement moves the addresses of the elements of a list variable into the address variables in the destination list. Each list variable element is placed successively into an address variable. If the source list variable contains an array or another list, the array or list is considered to be a single element. If the destination list contains address variable arrays, each element of the array is considered to be a single element.

Variable types that do not match cause the corresponding destination variable to be set to invalid. Typeless address variables match any type of variable.

updatab

label **`updatab`** *file* `;` *list*
label **`updatab`** *ifile* `;` *list*
label **`updatab`** *afile* `;` *list*

file is the label of a file declaration
ifile is the label of an ifile declaration
afile is the label of an afile declaration
list is the list of variables, literals, and tab control codes

Flags affected: none

See also: update

The updatab statement replaces parts of the last record read or written with characters from the list of variables. This operation cannot be performed on compressed files. An error will occur if an updatab statement is executed before any record has been read or written or if updatab is executed immediately after a delete instruction.

The list may contain tab control codes. A tab control is specified by **`*tab=`***n* or **`*`***n* , where *n* is an integer decimal constant or numeric variable. Values of *n* up to 65500 are allowed.

update

label **`update`** *file* `;` *list*
label **`update`** *ifile* `;` *list*
label **`update`** *afile* `;` *list*

file is the label of an file declaration
ifile is the label of an ifile declaration
afile is the label of an afile declaration
list is the list of variables, literals, and tab control codes

Flags affected: none

The update statement causes the last record read or written to be replaced with the logical record in the list. This operation cannot be done on compressed files. An error will occur if an update is executed before any record has been read or written or if an update is executed immediately after a delete instruction.

The list may contain tab control codes. A tab control is specified by **`*tab=`***n* or **`*`***n* , where *n* is an integer decimal constant or numeric variable. Values of *n* up to 65500 are allowed. The update statement will function in the same manner as updatab when tab controls are contained in the list.

wait

label **`wait`**
label **`wait`** *list*

list is a comma delimited list of queue variables and communication files

Flags affected: none

The wait statement without an operand pauses program execution until any queue is non-empty or any open communication file operation is in the "complete" or "error" state. Execution then continues with the statement after the wait statement.

The wait statement with a list pauses program execution until any queue specified in the list is non-empty or any open communication file specified in the list is in the "complete" or "error" state. Program execution then continues with the statement after the wait statement.

weof

label **`weof`** *file*`,` *numvar*
label **`weof`** *ifile*`,` *numvar*
label **`weof`** *afile*`,` *numvar*

file is the label of a file declaration
ifile is the label of an ifile declaration
afile is the label of an afile declaration
numvar is the second operand

Flags affected: none

The weof statement causes an end-of-file mark to be written to the file associated with the first operand. The weof statement causes the file to be truncated. The weof statement is considered to be random if the second operand is nonnegative. The weof statement is considered to be sequential if the second operand is negative.

If the weof statement is sequential and the second operand has a value of -3, the weof statement is ignored. If the second operand has a value of -1 or -2, then the end-of-file is set to the sequential data file position.

If the weof operation is random, then the end-of-file is set to the Nth record position of the file where N is the truncated value of *numvar*.

The weof statement is ignored for data type files and text type files that do not use an end-of-file character.

winrestore

label **`winrestore`** *charvar*

`winrest` and **`restore`** may be used in place of **`winrestore`**

charvar is the source operand

Flags affected: none

See also: winsave, winsize

The subwindow image stored in the source operand is displayed in the current subwindow. The source operand must have been filled by a prior winsave instruction. After the winrestore statement is executed, the cursor is positioned in the upper left corner of the subwindow. If the winrestore operation is unable to restore the subwindow image, an E 506 error occurs.

winsave

label **`winsave`** *charvar*

`save` may be used in place of **`winsave`**

charvar is the destination variable

Flags affected: eos

See also: winrestore, winsize

The characters and attributes displayed in the current subwindow are stored in the destination variable starting at the first character of the variable. The destination form pointer is set to one and the logical length pointer is set to the number of characters moved. The winsize statement can be used to determine the size of the destination variable. If the destination variable is not large enough to store all of the characters in the subwindow, then the eos flag is set. Otherwise, the eos flag is cleared.

winsize

label **`winsize`** *numvar*

numvar is the destination variable

Flags affected: none

See also: winsave

The winsize statement moves the number of characters needed to store the characters and attributes displayed in the current subwindow into the destination variable of the winsave statement.

write

label **`write`** *file* `,` *numvar* `;` *list*
label **`write`** *ifile* `,` *var* `;` *list*
label **`write`** *afile* `,` *numvar* `;` *list*
label **`write`** *afile* `;` *list*

`writab` may be used in place of **`write`**

file is the label of a file declaration
ifile is the label of an ifile declaration
afile is the label of an afile declaration
numvar is the second operand
var is the second operand
list is the list of variables, literals, and control codes

Flags affected: none

See also: format

The write statement writes the data contained in the list variables to the data file. The write is random if the second operand is a nonnegative numeric variable. The write is sequential if the second operand is a negative numeric variable. The write is indexed if the second operand is a character variable. The write is aimdexed when the fourth format of write statement is used.

Characters are written to a record in the data file from the variables and literals in the list.

If the variable from the list is a character variable, characters are written to the data file starting with the first character in the character variable through the character pointed to by the length pointer. Blanks are written for each character position after the length pointer through the maximum length of the variable. If the form pointer of the variable is zero, then blanks are written for each character position through the maximum length of the variable.

All characters in a numeric variable or a literal are written to the data file.

If the number of characters in the logical record to be written is greater than the record size specified in the file declaration, then the record is truncated before being written to the data file.

If the list is terminated with a semicolon (`;`), then no end-of-record character is written to the file. This is considered to be a partial write. The next write must be a sequential write. If the list is not terminated with a semicolon, then an end-of-record character is written. In any case, the file position is set to the character position after the last character written.

If the access is sequential and the second operand has a value of -3, then the file is positioned to the end of data file before the data is written. If the access is sequential and the second operand has a value other than -3, then data is written to data file starting at the current file position.

If the access is random, then the file position is set to the first character of the Nth record and the data is written to data file. N is the truncated value of the numeric variable that is the second operand.

If the access is indexed, then the logical string from the character variable that is the second operand is used as the new key to be put into the index. If duplicates are not allowed (i.e., **dup** is not in the ifile declaration), then the index is searched for a key that matches the new key. If a match is found, then the duplicate key error occurs. If no match is found or if duplicates are allowed, then the new key is inserted into the index and the record is written to the data file. The record is written at the file position where the end of file is before the operation is started. An end-of-file character is automatically written after the logical record is written. The file position is left pointing to the end-of-file character.

If the access is aimdexed, then the field information specified in the aimdex parameter list is used to update the information in the aimdex file. The record is written to the end of the data file in the same way as for an indexed write.

The operand list may contain tab control codes. A tab control is specified by ***tab=***n* or ******n* , where *n* is an integer decimal constant or numeric variable. Values of *n* up to 65500 are allowed. If tab controls are included, then the record being written must already exist because no end-of-record character is written to the file.

***sl** is the blank suppression control code. This control code affects all remaining character variables in the list or until an ***ll** or ***pl** control code is encountered. This control code causes no blanks to be written for the characters between the length pointer and the maximum length of the string. If the variable is cleared, then nothing is written and the write position remains unchanged.

***ll** is the logical length control code. This control code is in effect for all remaining character variables in the list or until an ***sl** or ***pl** control code is encountered. The logical string of the character variable is written to data file.

***pl** is the physical length control code. This control code cancels the effect of the ***ll** and ***sl** control codes.

format=charexp* is the format control code. This control code affects the next variable in the list. *charexp* is a format mask which is used to reformat numeric and character data. Refer to the explanation of the format statement for information about how the mask is used.

***zf** is the zero fill control code. This control code affects only the next variable in the list. If the next variable is a numeric variable, zeros are written instead of blanks from the variable. In addition, the minus sign (if it exists) is written as the first character from the variable.

***zs** is the zero suppress control code. This control code affects only the next variable in the list. If the next variable is numeric with a value of zero, blanks are written for each character position of the numeric variable (including the decimal point). If the variable is not numeric, it is written as usual.

***mp** is the minus over punch control code. This control code only affects the next variable in the list. If the next variable is a numeric variable and the value is negative, then the minus sign is replaced with a blank (or a zero if ***zf** is in effect) and the far right digit of the variable is altered before it is written to data file. The last character is altered to } if the far right digit was 0. The last character is altered to a letter between J and R if the far right digit was 1 through 9, respectively. If the runtime property **dbcdx.file.minusoverpunch=ascii** is specified, then p is used instead of } and q through y are used instead of J through R.

xor

label **`xor`** *charexp prep charvar*
label **`xor`** *hexcon prep charvar*
label **`xor`** *numexp prep numvar*

charexp is the source operand
hexcon is the source operand
numexp is the source operand
charvar is the destination operand
numvar is the destination operand

Flags affected: equal, eos

See also: and, or, not

For the first and second formats of the xor statement, the bitwise exclusive or operation is performed on the form pointed characters of the source and destination operands. The result is stored in the form pointed position in the destination operand. If a decimal, hexadecimal, or octal constant is used as an operand, the character represented by that character code is used. If the result is binary zero, the equal flag is set. If either string is null, the eos flag is set and no changes are made.

Fot the third format of the xor statement, the source and distination operands are converted to 32 bit integers and the xor operation is performed. The result is moved to the destination operand. If the result is zero, the equal flag is set. Otherwise, the equal flag is cleared.

The result of the xor statement is determined by comparing the bits in each operand:

0 XOR 0 evaluates to 0
0 XOR 1 evaluates to 1
1 XOR 0 evaluates to 1
1 XOR 1 evaluates to 0

GUI Programming Reference

Creation of Windows and Timers

Windows and timers are implemented as device variables. The prepare statement causes a window or timer to be created. Windows are shown when they are created.

The syntax of the prepare statement for creation of a window or timer device is:

`prepare` *device prep charexp*

The logical string of *charexp* contains a list of parameters separated by commas.

Parameters for Creation of a Window Device

window=*text*

window defines the name of the window device being created. *text* is the one to eight character window name. One of **window** or **floatwindow** must be the first parameter in the list.

floatwindow=*text*

floatwindow defines the name of the floating window device being created. *text* is the one to eight character window name. One of **window** or **floatwindow** must be the first parameter in the list. A floating window is always above regular windows, is fixed size and does not allow a menu or a toolbar.

pos=*horz* **:** *vert*

pos defines the position of the window relative to the upper left corner of the screen. *horz* is a horizontal position in pixels (one is the far left position). *vert* is a vertical position in pixels (one is the top position). If **pos** is omitted, the window is centered on the screen.

size=*horz-size* **:** *vert-size*

size defines the size of the window. *horz-size* is the horizontal size in pixels. *vert-size* is the vertical size in pixels. **size** must be specified for a floating window.

maximize

maximize specifies that the window will fill the primary display. The window title bar and system menu will be displayed. This keyword is ignored for at floating window.

title=*text*

text is the title of the window. The title will appear centered in the title bar.

noclose

noclose specifies that the window will not contain a close box or system command menu.

fixsize

fixsize defines the window to be of fixed size. Panning scroll bars will not be displayed.

noscrollbars

noscrollbars specifies that automatic scroll bar management will not be implemented in the window. All scrolling must be handled by the programmer. This keyword is ignored for at floating window.

notaskbarbutton

notaskbarbutton specifies that when the window is minimized, a button will not appear in the taskbar. The window can be restored programmatically using the restore change function. This keword is ignored for at floating window.

nofocus

nofocus specifies that the window will not be activated when it is created. This keyword is ignored for at floating window.

owner=*text*

owner specifies the name of the window that is the owner of this floatwindow. This keyword is only applicable to a floatwindow in Windows. When this is specified and the window that "owns" this window loses focus, this window is hidden. In addition, this window is always on top of the window that "owns" it, but is not on top of other windows.

pandlgscale

pandlgscale specifies that the window size will be scaled according to the **dbcdx.gui.pandlgscale** runtime property.

statusbar

statusbar specifies that the window will be created with an empty statusbar. This keyword is ignored for at floating window.

Parameters for the Creation of a Timer Device

timer=*text*

timer defines the name of the timer device being created. *text* is the one to eight character timer name.

Operation of Windows and Timers

The change statement

The syntax of the change statement is:

`change` *device* **,** *charexp* **;** *list*

The logical string of *charexp* contains the change function. The change values are contained in the *list*. The following change functions are valid for a window device:

change function = **"title"**

The window title will be changed to the value specified in *list*.

change function = **"size"**

The size of the window will be changed to the value specified in *list*. The format of list is *hhhhhvvvvv* where *hhhhh* is the horizontal size and *vvvvv* is the vertical size. Each value is five digits and may be blank filled on the left.

change function = **"position"**

The position of the window will be changed to the horizontal and vertical coordinates specified in *list*. The format of list is *hhhhhvvvvv* where *hhhhh* is the horizontal position and *vvvvv* is the vertical position.

change function = **"mouseon"**

All mouse movements will be reported by **POSN** messages.

change function = **"mouseoff"**

No mouse movements will be reported. This is the default state.

change function = **"statusbar"**

The text specified in *list* is displayed in the status bar at the bottom of the window. A status bar will be created if none is present.

change function = **"nostatusbar"**

The status bar is removed from the window.

change function = **"desktopicon"**

The icon in the upper left hand corner of the window will change to the icon resource named in *list* if the icon resource was defined as a 16 by 16 pixel icon. If the icon resource was defined as a 32 by 32 pixel icon, the large icon associated with the window will be changed.

change function = **"pointer"**

The cursor will change to the shape specified by the value in *list*. Valid values are **"appstarting"**, **"arrow"**, **"handpoint"**, **"wait"**, **"cross"**, **"ibeam"**, **"help"**, **"uparrow"**, **"sizeall"**, **"sizens"**, **"sizewe"**, **"sizenesw"**, **"sizenwse"**, **and "no"**.

change function = **"hscrollbarpos"**
The window's horizontal scroll bar is displayed and the position of the slider is specified by a five digit value *hhhhh* specified in *list*.

change function = **"hscrollbarrange"**
The range of the window's horizontal scroll bar is set to the values specified in list. The format of the values in list is *lllllhhhhhppppp*. *lllll* is the new low value of the range. *hhhhh* is the new high value of the range. *ppppp* is the new page size. Each value must be five characters long and may be filled with blanks on the left.

change function = **"hscrollbaroff"**
The horizontal scroll is removed from being displayed.

change function = **"vscrollbarpos"**
The window's vertical scroll bar is displayed and the position of the slider is specified by a five digit value *vvvvv* specified in *list*.

change function = **"vscrollbarrange"**
The range of the window's vertical scroll bar is set to the values specified in list. The format of the values in list is *lllllvvvvvppppp*. *lllll* is the new low value of the range. *vvvvv* is the new high value of the range. *ppppp* is the new page size. Each value must be five characters long and may be filled with blanks on the left.

change function = **"vscrollbaroff"**
The vertical scroll is removed from being displayed.

change function = **"focus"**
This window is made to be the active window.

change function = **"unfocus"**
This window is made to be inactive and the most recently active window is made to be the active window.

change function = **"careton"**
The text bar caret (a vertical bar) is displayed.

change function = **"caretoff"**
The text bar caret is removed from being displayed.

change function = **"caretsize"**
The height in pixels of the text bar is specified by the five digit value *nnnnn* in *list*.

change function = **"caretpos"**
The upper left corner of the text bar caret is specified by the horizontal and vertical values in list. The format of these values is *hhhhhvvvvv* where each value is 5 digits and each may be left filled with blanks.

change function = **"maximize"**
The window will become maximized.

change function = **"minimize"**
The window will become minimized.

change function = **"restore"**
A maximized or minimized window will be restored back to its normal size and position.

The following change functions are valid for a timer device:

change function = **"set"**
The timer is set. *list* contains a timestamp in the form *yyyymmddhhmmsspp*. If the trailing *sspp* or *pp* is omitted, it is assumed to be zero. When the time specified by the timestamp has passed, one tick message will be sent.

change function = **"clear"**
The timer is cleared.

change function = **"msetnnn"**

The timer is set to be a recurring (or multiple) timer. *nnn* is the number of seconds between tick messages. A timestamp may optionally be specified in *list*. If specified, the timestamp is in the same format as for the **"set"** change function. If timestamp is specified, the first tick message will occur when the timestamp has passed and additional tick messages will occur at *nnn* second intervals. If the timestamp is not specified, the first tick message will occur *nnn* seconds after the change function occurs.

The link statement

The syntax of the link statement is:

`link` *device prep queue*

The link statement links a queue to the window or timer specified by device.

The unlink statement

The syntax of the unlink statement is:

`unlink` *device*

The unlink statement destroys the link between the window specified by device and the queue to which it is linked.

Creation of Resources

The prepare statement with a resource variable creates menus, panels, dialog boxes, icons, toolbars, and special dialogs. The syntax of the prepare statement for creation of resources is:

`prepare` *resource prep charexp*

The logical string of *charexp* contains a list of parameters separated by commas.

Parameters for Creation of Menu and Popup Menu Resources

menu=*text*

menu specifies that the resource is a menu bar. *text* is the menu name (up to eight characters). menu must be the first parameter in the list of parameters.

popupmenu=*text*

popupmenu specifies that the resource is a popup menu. *text* is the menu name (up to eight characters). **popupmenu** must be the first parameter in the list of parameters.

main=*item* **:** *main-menu-entry*

main defines an entry on the menu bar. *item* is an integer value (from 1 to 32000) that is contained in the message sent to the queue linked with this menu resource. main-menu-entry must be a single word. At least one main parameter must precede all other parameters.

item=*item* **:** *text*

item defines an entry in a pulldown menu of a menu bar or an entry in a popup menu. **item** is an integer value (from 1 to 32000) that is contained in the message sent to the queue linked with this menu resource. *text* is the text string shown in the pulldown menu or popup menu entry. The text string ends when either a comma or the end of *charexp* is reached. A back slash (****) is the forcing character used to include a colon within the string.

iconitem=*item* **:** *icon* **:** *text*

iconitem is identical to **item** except in that it will display the named icon to the left of the text in the menu item.

checkitem=*item* **:** *text*

checkitem is identical to **item** with the additional ability to respond visually to the mark and unmark change commands.

submenu=*item* **:** *text*

submenu defines an entry in a pulldown menu or popup menu that causes another popup menu to be displayed to the right of the current menu. *item* is an integer value (from 1 to 32000) that is contained in the message sent to the queue linked with this menu resource. *text* is the text string shown in the popup menu. The text string ends when either a comma or the end of *charexp* is reached. A back slash (\) is the forcing character used to include a comma or a backslash within the string. item parameters that follow a submenu define entries in the submenu. Submenus may be nested up to five levels.

iconsubmenu=*item* **:** *icon* **:** *text*

iconsubmenu is identical to **submenu** except in that it will display the named icon to the left of the text in the submenu item.

endsubmenu

endsubmenu defines the end of a submenu. *item* parameters that follow an **endsubmenu** define entries in the menu that was active before the match submenu parameter.

gray

gray causes the preceding **main**, **item**, or **submenu** to be initially disabled.

key=*accelerator-key*

key defines the accelerator key that is associated with the preceding item. **key** may not be used in a popup menu resource. *accelerator-key* may be one of: **F1** through **F12**, **ALTF1** through **ALTF12**, **SHIFTF1** through **SHIFT12**, **CTRLF1** through **CTRLF12** **,CTRLSHIFTF1** through **CTRLSHIFTF12**, **CTRLA** through **CTRLZ** (may also be specified as **^A** through **^Z**), **CTRLSHIFTA** through **CTRLSHIFTZ**, **CTRLCOMMA**, **CTRLPERIOD**, **CTRLBSLASH**, **CTRLFSLASH**, **CTRLSEMICOLON**, **CTRLQUOTE**, **CTRLLBRACKET**, **CTRLRBRACKET**, **CTRLMINUS**, **CTRLEQUAL**, **PGUP**, **PGDN**, **INSERT**, **DELETE**, **HOME**, and **END**.

line

line causes a separator line to be displayed between item parameters.

Parameters for Creation of a Panel or Dialog Box Resource

For the creation of a control box resource, the first parameter specified in *charexp* must be panel or dialog.

Throughout this section, the units for all horizontal and vertical positions and dimensions are pixels.

The position of any control is with respect to the upper left corner of the panel or dialog, or if contained within a tab group, with respect to the upper left corner of the tab group. The horizontal and vertical positions are the position of the upper left corner of the control.

atext=*item* **:** *text*

atext defines an active text control. It is the same as a **vtext** control.

box=*horz-size* **:** *vert-size*

box defines a rectangular frame control that is a single pixel wide. *horz-size* and *vert-size* specify the size of the box.

boxtitle=*text* **:** *horz-size* **:** *vert-size*

boxtitle defines a box with a title. *text* is the text string displayed. The text string ends when either a comma or the end of *charexp* is reached. A back slash (\) is the forcing character used to include a colon within the string. *horz-size* and *vert-size* specify the size of the box.

boxtabs=*nnc* **:** *nnc* **:** *...* **:** *nnc*

boxtabs applies to the previous **listbox**, **listboxhs**, **mlistbox**, **mlistboxhs** or **dropbox** control. Up to 50 *nnc* values may be specified. Each *nn* value represents the width, in pixels, of each column in the **listbox**, **listboxhs**, **mlistbox**, **mlistboxhs** or **dropbox**. The *c* is optional and specifies justification. If omitted then left justification is assumed. The allowed values for *c* are **L** for left, **C** for center, and **R** for right justification. Each tab character in the line specifies a new column.

button=*item* **:** *text*

button defines a radio button control. item is an integer value (from 1 to 32000) that is contained in the message sent to the queue linked with this resource. text is the text string displayed to the right of the radio button. The text string ends when either a comma or the end of *charexp* is reached. A back slash (\) is the forcing character used to include a comma within the string. The **button** keyword must be specified between the **buttongroup** and **endbuttongroup** parameters.

buttongroup

buttongroup defines the start of a group of radio buttons. Only one radio button in a button group can be on at any time. **h**, **v**, **ha**, **helptext**, **va**, and **button** are the only parameters allowed between a **buttongroup** and **endbuttongroup** parameter.

catext=*item* **:** *text*

c**atext** defines an active centered text control. It is the same as a c**vtext** control, except that the control acts as a button so that a **PUSH** message is sent when the text is clicked.

cdropbox=*item* **:** *width* **:** *vert-size*

cdropbox may only appear between the **table** and **tableend**. **cdropbox** specifies a common dropbox column. *item* is the item number associated with the column. *width* is the width of the column in pixels. *vert-size* is the vertical size of the dropbox when opened. Each cell of the column has the same dropbox contents, but each cell in the column has its own selected text.

checkbox=*item* **:** *text*

checkbox defines a check-mark box. *item* is an integer value (from 1 to 32000) that may be used to change that status of the control and is contained in the message sent to the queue linked with this resource. *text* is the text string displayed to the right of the check-mark box. The text string ends when either a comma or the end of *charexp* is reached. A back slash (****) is the forcing character used to include a colon within the string.

checkbox=*item* **:** *width*

This form of the **checkbox** keyword must appear between **table** and **tableend** keywords. It defines a checkbox style column. *item* is the item number associated with the column. *width* is the width of the column in pixels.

ctext=*text* **:** *horz-size*

ctext causes text to be displayed centered within the area whose upper left corner is at the current control position and whose width in pixels is specified by *horz*-size. In all other respects, **ctext** is the same as **text**.

cvtext=*item* **:** *text* **:** *horz-size*

cvtext is functionally equivalent to the combination of **ctext** and **vtext**.

cvtext=*item* **:** *width*

This form of the **cvtext** keyword must appear between **table** and **tableend** keywords. It defines a column functionally equivalent to the combination of **ctext** and **vtext**.

defpushbutton=*item* **:** *text* **:** *horz-size* **:** *vert-size*

defpushbutton defines the default push button control. The default push button is considered pressed whenever the Enter key is pressed, unless another **pushbutton** has the keyboard focus. item is an integer value (from 1 to 32000) that is contained in the message sent to the queue linked with this resource. *text* is the text string displayed in the push button. The text string ends when either a comma or the end of *charexp* is reached. A back slash (****) is the forcing character used to include a colon within the string. *horz-size* and *vert-size* specify the size of the button.

dialog=*text*

dialog defines a modal dialog box. When the show statement is executed for this resource, keyboard and mouse focus is kept by the modal dialog until the hide statement is executed. The dialog box is displayed in front of all other windows. It is moveable. *text* is the resource name (up to eight characters). The text ends when either a comma or the end of *charexp* is reached. The dialog parameter may only be specified once and must be the first parameter.

dropbox=*item* **:** *horz-size* **:** *vert-size*

dropbox defines a drop box control. *item* is an integer value (from 1 to 32000) that is contained in the message sent to the queue linked with this resource. *horz-size* is the horizontal size. *vert-size* is the vertical size. If this keyword appears between **table** and **tableend** then it defines a dropbox style column. In that case *horz-size* defines the column width. Each cell in a dropbox style column has an independent set of contents in the drop down list. That is, the contents of the drop down list can be different for each cell in the column. If each cell in the column should have the same contents then use a cdropbox style column.

edit=*item* **:** *text* **:** *horz-size*

edit defines a text entry control. *item* is an integer value (from 1 to 32000) that is contained in the message sent to the queue linked with this resource. *text* is the text string initially displayed in the edit control field. The text string ends when either a comma or the end of *charexp* is reached. A back slash (****) is the forcing character used to include a colon within the string. *horz-size* is the horizontal size of the control.

edit=*item* **:** *width*

This form of the **edit** keyword must appear between **table** and **tableend** keywords. It defines an edit box style column. *item* is the item number associated with the column. *width* is the width of the column in pixels.

endbuttongroup

endbuttongroup defines the end of a group of radio buttons.

escpushbutton=*item* **:** *text* **:** *horz-size* **:** *vert-size*

escpushbutton is the escape or cancel push button control. It is exactly the same as a **pushbutton** except that it will also be considered to be pushed when the Esc key is pressed.

fedit=*item* **:** *text* **:** *text* **:** *horz-size*

fedit defines a formatted edit box control. It is the same as an **edit** control, except the first *text* field contains a string of characters that controls the entry and display of characters in the fedit control. Characters other than special format mask characters are displayed as is. Special format mask characters are: **A**, **U**, **L**, **Z**, **9**, **.** (period), and **** (backslash). **A** means allow any character to be entered. **U** and **L** mean allow any character, but force them to upper and lower case, respectively. **Z** and **9** mean only allow digits and right justify automatically when the control loses focus. When the control loses focus, any characters corresponding with the **9** mask character are replaced with **0** and any characters corresponding with the **Z** mask character are replaced with blank. The period character is also a special mask character which acts with the **9** and **Z** mask characters to allow for numeric data entry. A backslash (****) is the forcing character.

font=*fontname* **(***options***)**

font defines the font for text in the controls that follow. *fontname* and **(***options***)** are optional, but one or both must be specified. *fontname* is the font name. Valid font names are: **Courier**, **Helvetica**, **Times**, **Monaco**, **System**, and **Default**. *options* contains the font size and other font at tributes. The maximum number of different fonts allowed in panel and dialog resources is 30.

gray

gray specifies that the previous control is initially displayed in the gray (or inactive) state.

h=*horz*

The **h** parameter changes the horizontal position for the next control. *horz* is the horizontal position (one is the far left position).

ha=*horz-adj*

The **ha** parameter changes the horizontal position for the next control. *horz-adj* is the horizontal adjustment (positive or negative) of the horizon position.

helptext=*text*

helptext specifies the help text for the previous control.

hscrollbar=*item* **:** *horz-size* **:** *vert-size*

hscrollbar defines a horizontal scroll bar control. *item* is an integer value (from 1 to 32000) that is contained in the message sent to the queue linked with this resource. *horz-size* is the horizontal size. *vert-size* is the vertical size. The default range is one.

icon=*image*

icon specifies an icon. *image* is the name of an icon resource that contains the image that is displayed.

icondefpushbutton=*item* **:** *image* **:** *horz-size* **:** *vert-size*

icondefpushbutton defines a control that is the same as a **defpushbutton**, except that an icon is displayed inside of the button instead of text. If the size of the icon is larger than the button size, extra pixels are chopped off. If it is smaller, the background color will show.

iconescpushbutton=*item* : *image* : *horz-size* : *vert-size*

iconescpushbutton defines a control that is the same as a **escpushbutton**, except that an icon is displayed inside of the button instead of text. If the size of the icon is larger than the button size, extra pixels are chopped off. If it is smaller, the back ground color will show.

iconpushbutton=*item* : *image* : *horz-size* : *vert-size*

iconpushbutton defines a control that is the same as a pushbutton, except that an icon is displayed inside of the button instead of text. If the size of the icon is larger than the button size, extra pixels are chopped off. If it is smaller, the background color will show.

insertorder

insertorder applies to the previous **listbox**, **listboxhs**, **mlistbox**, **mlistboxhs** or **dropbox** control. It means that the order of the entries in the **listbox**, **listboxhs**, **mlistbox**, **mlistboxhs** or **dropbox** is determined by the order and location in which each entry was inserted. See the insert change statement operations. If this keyword appears between **table** and **tableend** then it applies to the most recent **dropbox** or **cdropbox** column. Note that this attribute always applies to all rows in a column.

justifycenter

justifycenter specifies that the previous edit, ledit or fedit control should initially have its text center justified. If this keyword appears between **table** and **tableend** then it applies to the most recent edit column.

justifyleft

justifyleft specifies that the previous edit, ledit or fedit control should initially have its text left justified. This is the default. If this keyword appears between **table** and **tableend** then it applies to the most recent edit column.

justifyright

justifyright specifies that the previous edit, ledit or fedit control should initially have its text right justified. If this keyword appears between **table** and **tableend** then it applies to the most recent edit column.

ledit=*item* : *text* : *horz-size* : *max-chars*

ledit defines a defined character maximum edit control. It is the same as an edit control, except that *max-chars* specifies the maximum number of characters that may entered into the control.

listbox=*item* : *horz-size*:*vert-size*

listbox defines a single line selectable list box control. *item* is an integer value (from 1 to 32000) that is contained in the messages sent to the queue linked with this resource. *horz-size* is the horizontal size. *vert-size* is the vertical size.

listboxhs=*item* : *horz-size* : *vert-size*

listboxhs defines a single line selectable list box, which has a horizontal scroll bar that automatically appears when the width of a line of text is greater than the width of the listboxhs control, or when the sum of boxtab widths for the control is greater than the width of the control. *item* is an integer value (from 1 to 32000) that is contained in the messages sent to the queue linked with this resource. *horz-size* is the horizontal size. *vert-size* is the vertical size.

ltcheckbox=*item* : *text*

ltcheckbox defines a check-mark box. It is the same as a checkbox control except that the text string is displayed to the left of the check-mark box.

medit=*item* : *text* : *horz-size* : *vert-size*

medit defines a multiple line edit control. It is the same as an edit control, except that the *horz-size* and *vert-size* specify the size of the multiple line edit box and word wrap occurs at the right side of each line.

mediths=*item* : *text* : *horz-size* : *vert-size*

mediths is an **medit** control with a horizontal scrollbar.

medits=*item* : *text* : *horz-size* : *vert-size*
: **medits** is an **medit** control with a horizontal and vertical scrollbar.

meditvs=*item* : *text* : *horz-size* : *vert-size*
: **meditvs** is an **medit** control with a vertical scrollbar.

mledit=*item* : text : horz-size:vert-size : max-chars
: **mledit** defines a defined character maximum multiple line edit control. It is the same as an **medit** control, except that *max-chars* specifies the maximum number of characters that may entered into the control.

mlediths=*item* : *text* : *horz-size* : *vert-size* : *max-chars*
: **mlediths** is an **mledit** control with a horizontal scrollbar.

mledits=*item* : *text* : *horz-size* : *vert-size* : *max-chars*
: **mledits** is an **mledit** control with a horizontal and vertical scrollbar.

mleditvs=*item* : *text* : *horz-size* : *vert-size* : *max-chars*
: **mleditvs** is an **mledit** control with a vertical scrollbar.

mlistbox=*item* : *horz-size* : *vert-size*
: **mlistbox** defines a multi-line selectable list box. *item* is an integer value (from 1 to 32000) that is contained in the messages sent to the queue linked with this resource. *horz-size* is the horizontal size. *vert-size* is the vertical size.

mlistboxhs=*item* : *horz-size* : *vert-size*
: **mlistboxhs** defines a multi-line selectable list box, which has a horizontal scroll bar that automatically appears when the width of a line of text is greater than the width of the **mlistboxhs** control, or when the sum of boxtab widths for the control is greater than the width of the control. *item* is an integer value (from 1 to 32000) that is contained in the messages sent to the queue linked with this resource. *horz-size* is the horizontal size. *vert-size* is the vertical size.

noclose
: **noclose** causes there to be no close button in the dialog box. This parameter may only be used with dialog.

nofocus
: **nofocus** specifies that the previous control cannot receive keyboard focus.

noheader
: **noheader** specifies that the table does not have a header row. This keyword must be specified between the **table** and **tableend** keywords. Note that the **table** keyword must still specify the titles of the columns, which are ignored, to define number of columns in the table.

panel=*text*
: **panel** defines a clipped box without borders or title that contains controls. It is attached to a window when the show statement is executed. *text* is the resource name (up to eight characters). The panel parameter may only be specified once and must be the first parameter.

pedit=*item* : *text* : *horz-size*
: **pedit** defines a password edit box. A password edit box control is the same as an **edit** control, except that an asterisk or block is displayed for each character in the control.

pledit=*item* : *text* : *horz-size* : *max-chars*
: **pledit** defines a maximum character password edit box. It is the same as an **edit** control, except that *max-chars* specifies the maximum number of characters that may be entered into the control, and an asterisk or block is displayed for each character in the control.

popbox=*item* : *horz-size*
: **popbox** defines a control that is a combination of a readonly edit box and a pushbutton. *item* is an integer value (from 1 to 32000) that identifies the control. *horz-size* is the horizontal size of the control. The contents of this control are changed using the text change command. If the user clicks anywhere in the control a **PUSH** message is put in the queue. This control can receive the

focus. If this keyword appears between **table** and **tableend** then it defines a popbox style column. In that case *horz-size* defines the column width.

progressbar=*item* **:** *horz-size* **:** *vert-size*

progressbar defines a progress bar control. *item* is an integer value (from 1 to 32000) that is contained in the message sent to the queue linked with this resource. *horz-size* is the horizontal size. *vert-size* is the vertical size. The default range is 1 to 100.

pushbutton=*item* **:** *text* **:** *horz-size* **:** *vert-size*

pushbutton defines a pushbutton control. *item* is an integer value (from 1 to 32000) that is contained in the message sent to the queue linked with this resource. *text* is the text string displayed in the push button. The text string ends when either a comma or the end of *charexp* is reached. A back slash (\) is the forcing character used to include a colon within the string. *horz-size* and *vert-size* specify the size of the button.

readonly

readonly specifies that the previous control will initially be a read only control. The previous control must be an **edit** control, or a table if **readonly** is immediately after a **tableend**. Controls in the readonly state receive keyboard focus, but may not be changed by the user. The readonly and showonly states are the same, except that a control in the showonly state never receives keyboard focus.

ratext=*item* **:** *text*

ratext defines an active right justified text control. It is the same as an **rvtext** control, except that the control acts as a button so that a **PUSH** message is sent when the text is clicked.

rtext=*text*

rtext causes text to be displayed such that the upper right corner of the area used by the text is the current control position. In all other respects, **rtext** is the same as **text**.

rvtext=*item* **:** *text*

rvtext is functionally equivalent to the combination of **rtext** and **vtext**.

rvtext=*item* **:** *width*

This form of the **rvtext** keyword must appear between **table** and **tableend** keywords. **rvtext** defines a static text column. *item* is the item number associated with the column. *width* is the width of the column in pixels. Text in this cell will be right justified.

showonly

showonly specifies that the previous control, or table if immediately after a **tableend,** will not respond to mouse clicks and cannot receive keyboard focus. **showonly** does not apply to popbox controls.

size=*horz-size*:*vert-size*

size defines the size of the control box. *horz-size* is the horizontal size and *vert-size* is the vertical size of the dialog box. This parameter is ignored for a panel.

tab=*item*:*text*

tab specifies the start of a new page of controls in a tab group. *item* is the item number associated with this page. *text* is the description of this page.

tabgroup=*horz-size*:*vert-size*

tabgroup specifies the start of a multi-page group of controls. **tabgroup** must be followed immediately by the first tab specifier. Two or more tab specifiers are each followed by the controls that are contained within the page of controls for that tab. The group ends with **tabgroupend**. The size of the area containing the group controls is specified by *horz-size* and *vert-size*. Within a tab group, the horizontal and vertical positions of each control (specified by *h*, *v*, *ha* and *va*) are with respect to the upper left corner of the display area of the tab group.

tabgroupend

tabgroupend specifies the end of a group of controls.

table=*item* **:** *horz-size* **:** *vert-size* **:** *title-text* **:** … **:** *title-text*

table specifies the start of a group of controls comprising a table. *item* is the item number associated with this table. *horz-size* is the horizontal size. *vert-size* is the vertical size. *title-text* may be repeated one or more times. Each *title-text* is a column heading. The table will have a number of columns equal to the number of times that *title-text* appears. Scroll bars on the right and bottom will appear as needed. To embed a colon or comma in a column title, precede it with a backslash. A table must have at least one column.

tableend

tableend specifies the end of the control group for a table.

textcolor=*color*

textcolor defines the color for the text in the controls that follow. *color* may be one of: **red**, **green**, **blue**, **cyan**, **yellow**, **magenta**, **black**, or **white**. Note that the color of a pushbutton text cannot be changed and will not be affected by this parameter.

text=*text*

text causes text to be displayed. *text* is the text string displayed. The text string ends when either a comma or the end of *charexp* is reached. A back slash (\) is the forcing character used to include a comma within the string.

title=*text*

title defines the title of a modal dialog box. *text* is a text string that is the title. The text string ends when either a comma or the end of *charexp* is reached. A back slash (\) is the forcing character used to include a comma within the string. This parameter may only be specified once and may not be used with panel.

tree=*item* **:** *horz-size* **:** *vert-size*

tree defines a control that displays a hierarchical list of items. The tree control is initially empty with an insert position at the root level.

v=*vert*

The **v** parameter changes the vertical position for the next control. *vert* is the vertical position (one is the top position).

va=*vert-adj*

The **va** parameter changes the vertical position for the next control. *vert-adj* is the vertical adjustment (positive or negative) of the vertical position.

vicon=*item* **:** *image*

vicon specifies a variable icon. *item* is an integer value (from 1 to 32000) that will be used in the change statement to change the icon displayed. *image* is the name of an icon resource that contains the image that is displayed initially.

vscrollbar=*item* **:** *horz-size* **:** *vert-size*

vscrollbar defines a vertical scroll bar control. *item* is an integer value (from 1 to 32000) that is contained in the message sent to the queue linked with this resource. *horz-size* is the horizontal size. *vert-size* is the vertical size. The default range is one.

vtext=*item* **:** *text*

vtext is the same as a text control, except the text value may be changed with the change statement. item is an integer value (from 1 to 32000) that identifies the control.

vtext=*item* **:** *width*

This form of the **vtext** keyword must appear between **table** and **tableend** keywords. **vtext** defines a static text column. *item* is the item number associated with the column. *width* is the width of the column in pixels.

Parameters for Creation of an Icon Resource

Icons created for use in a tree must be 16 pixels wide and 16 pixels high.

icon=*text*

icon specifies that the resource is an icon. *text* is the icon name (up to eight characters). **icon** must be the first parameter in the list of parameters.

h=*horz-size*

h defines the horizontal size of the icon. The size is measured in pixels. *horz-size* is an integer with valid values from 4 through 64. If **h** is not specified, the horizontal size of the icon is 16.

v=*vert-size*

v defines the vertical size of the icon. The size is measured in pixels. *vert-size* is an integer with valid values from 4 through 64. If **v** is not specified, the vertical size of the icon is 16.

colorbits=*nn*

colorbits defines the number of color bits per pixel for this icon. The valid values for *nn* are **1**, **4**, and **24**. If **colorbits** is not specified, the icon is a black and white icon (the default is **1**).

pixels=*xxxx*...

pixels defines the color values of each pixel in the icon. Each *x* is a hexadecimal character (values **0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**, **A**, **B**, **C**, **D**, **E**, **F**). The pixels are specified left to right and top to bottom. For **colorbits=1** and **colorbits=4**, each hexadecimal character represents a pixel. For **colorbits=1**, if the hexadecimal character is zero, the pixel is black. Otherwise the pixel is white. For **colorbits=4**, here is a list of what each hexadecimal character represents:

value 0 is black
value 1 is dark blue
value 2 is dark green
value 3 is dark cyan
value 4 is dark red
value 5 is dark magenta
value 6 is brown
value 7 is light gray
value 8 is dark gray
value 9 is bright blue
value A is bright green
value B is bright cyan
value C is bright red
value D is bright magenta
value E is yellow
value F is white

For **colorbits=24**, each six hexadecimal characters represent the color of one pixel. From each group of six characters, the first two represent the blue intensity, the second two represent the green intensity, and the third two characters represent the red intensity. In each pair, the first character is the high order hexadecimal digit and the second is the low order hexadecimal digit. The character **T** can be used to represent a pixel that should be transparent in icons with **colorbits=1** or **colorbits=4**. For **colorbits=24** icons, use **TTTTTT** to define a transparent pixel.

Parameters for Creation of a Toolbar Resource

toolbar=*text*

toolbar specifies that the resource is a toolbar. *text* is the resource name (up to eight characters). **toolbar** must be the first parameter in the list of parameters.

pushbutton=*item* **:** *icon* **:** *text*

pushbutton defines a push button control. *item* is an integer (from 1 to 32000) that is contained in the message sent to the queue linked with this resource. *icon* is the name of the icon resource that contains the image that is displayed. *text* is the help or tooltip text that describes the button.

lockbutton=*item* **:** *icon* **:** *text*

lockbutton defines a push button control. *item* is an integer (from 1 to 32000) that is contained in the message sent to the queue linked with this resource. *icon* is the name of the icon resource that contains the image that is displayed. *text* is the help or tooltip text that describes the button.

dropbox=*item* **:** *horz-size* **:** *vert-size*

dropbox defines a drop box control. *item* is an integer value (from 1 to 32000) that is contained in the message sent to the queue linked with this resource. *horz-size* is the horizontal size. *vert-size* is the vertical size.

gray
gray specifies that the previous button is initially displayed in the gray (or inactive) state.

space
space defines a separator between buttons.

Parameters for Creation of Special Dialogs

opendirdlg=*text*
openfiledlg specifies that a standard directory chooser dialog resource is created. *text* is the resource name (up to eight characters). This must be the first parameter in the list of parameters.

openfiledlg=*text*
openfiledlg specifies that a standard file chooser dialog resource is created. *text* is the resource name (up to eight characters). This must be the first parameter in the list of parameters.

saveasfiledlg=*text*
saveasfiledlg specifies that a standard save file as dialog resource is created. text is the resource name (up to eight characters). This must be the first parameter in the list of parameters.

default=*text*
default specifies default directory or file that will be highlight when the special dialog is first shown. *text* specifies the directory or file. This parameter may follow the opendirdlg, openfiledlg, or saveasfiledlg parameter.

fontdlg=*text*
fontdlg specifies that a standard font chooser dialog resource is created. *text* is the resource name (up to eight characters). This must be the only parameter in the list of parameters.

colordlg=*text*
colordlg specifies that a standard color chooser dialog resource is created. *text* is the resource name (up to eight characters). This must be the only parameter in the list of parameters.

alertdlg=*text*
alertdlg specifies that an alert message box dialog resource is created. *text* is the resource name (up to eight characters). One **text** parameter follows this keyword.

text=*text*
text specifies the text that will be displayed in an alert dialog.

Operation of Resources

The show statement

The syntax of the show statement is:
`show` *resource prep device prep numexp prep numexp*

The show statement causes the specified resource to be displayed in the window specified by *device*. The optional third and fourth operands specify the horizontal and vertical positions of the upper left corner of the resource. For dialogs, this is relative to the screen. For panels and popupmenus this is relative to the window. The position operands are ignored for other resources. Only one toolbar may be shown at any given time on a window.

If the third and fourth operands are not specified, a dialog is displayed at the center of the screen, panels popupmenus are displayed at position **1:1** with respect to the window. For dialogs only, if either position operand is negative, the dialog will be centered on the screen. Negative positions are meaningful for popupmenus and panels.

The hide statement

The syntax of the hide statement is:
`hide` *resource*

The hide statement erases the specified resource from the window in which it is shown.

The link statement

The syntax of the link statement is:

`link` *resource prep queue*

The link statement links a queue with the menu or a control box.

The unlink statement

The syntax of the unlink statement is:

`unlink` *resource*

The unlink statement destroys the link between the specified resource and queue with which it is linked.

The query statement

The syntax of the query statement is:

`query` *resource*`,` *charexp*`;` *list*

The logical string of *charexp* is the query function. The result returned by a query statement is a string of characters that is moved to the list of variables in the same manner as for a read statement.

query function = **"status***nnnnn***"** (return the status, *nnnnn* is a one to five-digit item number)

- For check-mark boxes, radio buttons and menu items the result is *cs* where:
  - *c* is **Y** or **N** (checked status), always **N** if the item is in a toolbar
  - *s* is **G** or **A** (available status)
- For edit fields the result is *text*
- For scroll bars the result is *nnnnn* which is the current position
- For trees the result is the text of selected entry
- For list boxes and drop boxes the result is text of selected entry, unless **lineon** is in effect the result moved into the first variable in list will be the one-based line number of the selection. The receiving variable can be numeric in this case. A zero will be returned if there is no selection.
- For multi-line list boxes the result is a list which is comma-delimited concatenation of the text of all selectedlines, unless **lineon** is in effect the result moved into the first variable in list will be the comma delimited, one-based line numbers of all selected lines.
- For toolbar resources the result is *c* where:
  - *c* is **Y** or **N** (lock button locked status)

query function = **"status***nnnnn***[***row***]**"

For a table this function queries the status of the cell at row and column. *nnnnn* is the item number associated with the column. *row* is a one-based index of the row. This query will return the text of a **popbox**, **edit**, **vtext**, or **rvtext** cell in list. This query will return **Y** or **N** for a checkbox cell in list. For a **dropbox** or **cdropbox** cell *row* is required and this query will will return the selected text, unless **lineon** is in effect then the returned value will be the one-based index of the selected line instead of the text. If nothing is selected, and **lineon** is in effect, then a zero will be returned. If nothing is selected, and **lineon** is not in effect, then a zero length string will be returned.

query function = **"getrowcount***nnnnn***"**

For a table this query will return the five digit count of rows in the table.

The change statement

The syntax of the change statement is:

change *resource*`,` *charexp*`;` *list*

The logical string of *charexp* is composed of one or more change functions, each of which may be followed by zero or more items numbers. The functions and the item numbers must be separated by commas except for the first item number after a function. The comma is optional there. Item numbers may have leading blanks. The data field (*list*) will be used by all of the functions in a particular change that require data.

In all the following change functions, unless otherwise specified, *nnnnn* represents the item number of a control. Item numbers can be one to five digits in length. Valid colors are one of: **red**, **green**,**blue**, **cyan**, **yellow**, **magenta**, **black**, or **white,** or a numeric value which is interpreted as a 24 bit RGB value.

change function = **"adddropbox***nnnnn***"** (add dropbox)
For a toolbar, add a new dropbox control to a toolbar. *list* contains a string of 10 characters in the format *hhhhhvvvvv* where *hhhhh* is the horizontal size of the dropbox and *vvvvv* is the vertical size

change function = **"addlockbutton***nnnnn***"** (add lockbutton)
For a toolbar, *list* contains the name of the icon resource for the new lock button

change function = **"addpushbutton***nnnnn***"** (add pushbutton)
For a toolbar, *list* contains the name of the icon resource for the new push button

change function = **"addrow***nnnnn***"** (add row)
For a table, add an empty row to the bottom of the table. *nnnnn* is the item number of the table.

change function = **"addspace"** (add space)
For a toolbar, add a space before or after the next button

change function = **"available***nnnnn***"** (make control available)
For a menu item, toolbar button, or any panel or dialog control, make control available.

change function = **"availableall"**
For a panel or dialog this will make all controls available.
For a menu, all menu items are made available.

change function = **"collapse***nnnnn***"**
For a tree control, *list* contains the line number at the current indent level that will be collapsed to hide the children. If the line number is invalid, an I794 error will occur.

change function = **"color"** (set color)
For a color picker dialog, set the default color specified in *list*. The format of the color is this 15 character string of digits *rrrrrgggggbbbbb* where *rrrrr* is the red intensity, *ggggg* is the green intensity, and *bbbbb* is the blue intensity.

change function = **"defpushbutton***nnnnn***"** (new default pushbutton)
For a pushbutton, make the pushbutton specified by *nnnnn* be the default pushbutton.

change function = **"delete***nnnnn***"** (delete a line)
For a listbox, listboxhs, mlistbox, mlistboxhs or dropbox control, *list* contains text of item that will be deleted.
For a toolbar button, delete button whose item number is *nnnnn*
For a tree control, delete the current line from the control. If the current line is invalid an I794 error will occur. The delete function invalidates the current position, but the indent level will remain unchanged.
For a cdropbox column in a table, where *nnnnn* is the item number associated with the column, *list* contains the text of the item that will be deleted.

change function = **"delete***nnnnn***[***row***]**" (delete a line from a dropbox in a table cell)
For a dropbox column in a table, *nnnnn* is the item number associated with the column. *row* is a one-based index of the row. *list* contains the text of the item that will be deleted.

change function = **"deleteallrows***nnnnn***"** (delete all rows in a table)
For a table, delete all the rows. *nnnnn* is the item number associated with the table.

change function = **"deleteline***nnnnn***"** (delete a line by line number)
For a listbox, listboxhs, mlistbox, mlistboxhs or dropbox control, *list* contains the line number of the line that will be deleted.
For a cdropbox column in a table, where *nnnnn* is the item number associated with the column, *list* contains the line number of the line that will be deleted.

change function = **"deleteline***nnnnn***[***row***]"** (delete a dropbox item from a table cell)
For a table, and only for a dropbox column, *nnnnn* is the item number associated with the

column. *row* is a one-based index of the row. *list* contains the line number of the item that will be deleted.

change function = **"deleterow***nnnnn***[***row***]"** (delete a row from a table)
For a table, delete a row indexed by *row* (one-based) where *nnnnn* is the item number of the table.

change function = **"deselect***nnnnn***"** (deselect item)
For a multi-line list box, *list* contains the text of an item for which highlighting is turned off.
For a tree control, the line at the current position will be deselected if it is currently selected.

change function = **"deselectall***nnnnn***"** (deselect all items)
For a multi-line list box, highlighting is turned off for all lines.

change function = **"deselectline***nnnnn***"** (deselect item by line number)
For a multi-line list box, *list* contains the line number of the line to turn off highlighting for.

change function = **"dirname"** (set directory)
For an open directory dialog, set the default directory to the name specified in *list.*

change function = **"downlevel***nnnnn***"** (position down one level)
For a tree control, decrease the current indent level one towards the root level.

change function = **"erase***nnnnn***"** (erase all)
For an edit field, erase the text field.
For a listbox, listboxhs, mlistbox, mlistboxhs, dropbox, or tree control, delete all items.
For a vtext, rvtext, cvtext, or popbox control, erase the text.
For a cdropbox column in a table, *nnnnn* is the item number associated with the column. Remove all of the lines from the cdropbox.

change function = **"erase***nnnnn***[***row***]"** (erase all lines from a dropbox in a table)
For a text or dropbox column in a table, *nnnnn* is the item number associated with the column. *row* is a one-based index of the row. For a text cell, erase the text. For a dropbox cell, remove all lines from the dropbox.

change function = **"escpushbutton***nnnnn***"** (new escape pushbutton)
For a pushbutton, make the pushbutton specified by *nnnnn* be the escape pushbutton

change function = **"expand***nnnnn***"** (expand a tree)
For a tree control, *list* contains the line number at the current indent level that will be expanded to reveal the children. If the line number is invalid, an I794 error will occur.

change function = **"feditmask***nnnnn***"** (change mask)
For an fedit field, *list* contains the new mask string.

change function = **"filename"** (set filename)
For an open file or save as file dialog, set the default filename to the name specified in *list.*

change function = **"focus***nnnnn***"** (set focus)
The keyboard focus is switched to the control specified by *nnnnn*.

change function = **"focusoff"** (stop focus report)
This function stops **FOCS** messages from being sent by the dialog or panel.

change function = **"focuson"** (report focus change)
This function causes all focus changes with a dialog or panel to be reported with **FOCS** messages.

change function = **"fontname"** (set font name)
For a font picker dialog, set the default font name to the name specified in *list*.

change function = **"gray***nnnnn***"** (make control gray or unavailable)
For a menu item, toolbar button or any panel or dialog control, make the control gray.

change function = **"grayall"** (make all controls gray or unavailable)
For a panel or dialog, make all of the controls gray.
For a menu, all menu items are made gray.

change function = "**helptext***nnnnn*" (change help text)

For all panel, dialog, and toolbar controls, *list* contains the help text for the control specified.

change function = "**hide***nnnnn*" (hide menu entry)

For a menu item, hide the menu item.

change function = "**icon***nnnnn*" (set icon)

For a toolbar button, *list* contains the name of the icon resource to be displayed in the button.
For a tree control, list contains the name of the icon to be displayed at the current insert position.
For a variable icon control, *list* contains the name of the icon resource to display.
For an icon pushbutton, *list* contains the name of the icon resource to display. The icon contained in the icon resource must be the same size as the icon being replaced.

change function = "**insert***nnnnn*" (insert item)

For a listbox, listboxhs, mlistbox, mlistboxhs or dropbox control, *list* contains text of item that will be inserted.
For a tree control, the text in *list* will be inserted immediately after the line at the current position. If the line number is invalid, the insert function is the same as the insertafter function.
For a cdropbox column in a table, *nnnnn* is the item number associated with the column. *list* contains the text that will be inserted in the cdropbox.

change function = "**insert***nnnnn*[*row*]" (insert a line in a dropbox in a table)

For a dropbox column in a table, *nnnnn* is the item number associated with the column. *row* is a one-based index of the row. *list* contains the text that will be inserted in the dropbox.

change function = "**insertafter***nnnnn*" (insert item)

For a listbox, listboxhs, mlistbox, mlistboxhs or dropbox control, if *list* is not specified this causes future insert and minsert change functions to insert lines after the last entry in the list or drop box. If *list* is specified, this causes future insert and minsert change functions to insert lines after the entry whose text value matches *list.*
For a tree control, the text in *list* is inserted after the last line at the current indent level.
For a toolbar resource, if *nnnnn* is not specified this causes future addpushbutton, addlockbutton addropbox, and addspace change functions to insert buttons after the first button in the toolbar. If *nnnnn* is specified this causes future addpushbutton, addlockbutton, addropbox, and addspace change functions to insert buttons after the button whose item number is *nnnnn*.
For a cdropbox column in a table, *nnnnn* is the item number associated with the column. If *list* is not specified, this causes future insert and minsert change functions to insert lines after the last entry in the cdropbox. If *list* is specified, this causes future insert and minsert change functions to insert lines immediately after the entry whose text value matches *list*.

change function = "**insertafter***nnnnn*[*row*]" (insert in a dropbox in a table)

For a dropbox column in a table, *nnnnn* is the item number associated with the column. *row* is a one-based index of the row. If *list* is not specified, this causes future insert and minsert change functions to insert lines after the last entry in the dropbox. If *list* is specified, this causes future insert and minsert change functions to insert lines immediately after the entry whose text value matches *list*.

change function = "**insertbefore***nnnnn*" (insert item)

For a listbox, listboxhs, mlistbox, mlistboxhs or dropbox controls, if *list* is not specified this causes future insert and minsert change functions to insert lines before the first entry in the list or drop box. If *list* is specified this causes future insert and minsert change functions to insert lines immediately before the entry whose text value matches *list.*
For a tree control, the text in *list* is inserted before the first line at the current indent level.
For a toolbar resource if *nnnnn* is not specified this causes future addpushbutton, addlockbutton addropbox, and addspace change functions to insert buttons before the first button in the toolbar. If *nnnnn* is specified this causes future addpushbutton, addlockbutton, addropbox, and addspace change functions to insert buttons before the button whose item number is *nnnnn*.
For a cdropbox column in a table, *nnnnn* is the item number associated with the column. If *list* is not specified, this causes future insert and minsert change functions to insert lines before

the first entry in the cdropbox. If *list* is specified, this causes future insert and minsert change functions to insert lines immediately before the entry whose text value matches *list*.

change function = **"insertbefore***nnnnn*[*row*]**"** (insert in a dropbox in a table)
For a dropbox column in a table, *nnnnn* is the item number associated with the column. *row* is a one-based index of the row. If *list* is not specified, this causes future insert and minsert change functions to insert lines before the first entry in the dropbox. If *list* is specified, this causes future insert and minsert change functions to insert lines immediately before the entry whose text value matches *list*.

change function = **"insertlineafter***nnnnn*" (specify insert point)
For a listbox, listboxhs, mlistbox, mlistboxhs or dropbox control, future insert and minsert change functions will insert lines after the line whose number is specified in *list*.
For a cdropbox column in a table, *nnnnn* is the item number associated with the column. Future insert and minsert change functions will insert lines after the line whose line number is specified in *list*.

change function = **"insertlineafter***nnnnn*[*row*]**"** (specify insert point in a dropbox in a table)
For a dropbox column in a table, *nnnnn* is the item number associated with the column. *row* is a one-based index of the row. Future insert and minsert change functions will insert lines immediately after the line whose line number is specified in *list*.

change function = **"insertlinebefore***nnnnn***"** (specify insert point)
For a listbox, listboxhs, mlistbox, mlistboxhs or dropbox controls, Future insert and minsert change functions will insert lines immediately before the line whose number is specified in *list.*
For a cdropbox column in a table, *nnnnn* is the item number associated with the column. Future insert and minsert change functions will insert lines immediately before the line whose number is specified in *list.*

change function = **"insertlinebefore***nnnnn*[*row*]**"** (specify insert point in a dropbox in a table)
For a dropbox column in a table, *nnnnn* is the item number associated with the column. *row* is a one-based index of the row. Future insert and minsert change functions will insert lines immediately before the entry whose text value matches *list*.

change function = **"insertrowbefore***nnnnn*[*row*]**"** (insert an empty row in a table)
For a table, insert an empty row before the row indexed by *row* (one-based) for table with item number *nnnnn*.

change function = **"itemoff"** (no ITEM messages)
This function causes **ITEM** messages to be suppressed. This is the default state except for toolbars.

change function = **"itemon"** (send ITEM messages)
This function causes **ITEM** messages to be reported. This is the default state for toolbars.

change function = **"justifycenter***nnnnn***"** (set text justification)
For an edit field, center justify the text string.

change function = **"justifyleft***nnnnn***"** (set text justification)
For an edit field, left justify the text string.

change function = **"justifyright***nnnnn***"** (set text justification)
For an edit field, right justify the text string.

change function = **"lineoff"** (line numbers off)
This function causes all **ITEM** and **PICK** messages for listbox, listboxhs, mlistbox, mlistboxhs and dropbox controls to contain the text value of the lines. For tree controls, this function causes **OPEN**, **CLOS**, and **PICK** messages to return the text of the item.

change function = **"lineon"** (line numbers on)
This function causes all **ITEM** and **PICK** messages for listbox, listboxhs, mlistbox, mlistboxhs and dropbox controls to contain line numbers, not text. For tree controls, this function causes

OPEN, **CLOS**, **ITEM**, and **PICK** messages to return a comma-delimited list of positions at each level leading up to the selected item. For listboxes and dropboxes, the query status function will return the (one-based) line number of the selected item, not the text. A zero will be returned if there is no selection.

change function = **"locate**_nnnnn_**"** (locate item)

For a list box or drop box, *list* contains the text of the item to be highlighted.

For a cdropbox column in a table, *nnnnn* is the item number associated with the column. *list* contains the text of the line to be highlighted.

change function = **"locate**_column_**[**_row_**]"** (locate and highlight a line in a dropbox in a table cell)

For a dropbox column in a table, *nnnnn* is the item number associated with the column. *row* is a one-based index of the row. *list* contains the text of the line to be highlighted.

change function = **"locateline**_nnnnn_**"** (locate and highlight by line number)

For a list box or drop box, *list* contains the line number to be highlighted.

For a cdropbox column in a table, *nnnnn* is the item number associated with the column. *list* contains the the line number to be highlighted.

change function = **"locateline**_column_**[**_row_**]"** (locate and highlight a line in a dropbox in a table cell)

For a dropbox column in a table, *nnnnn* is the item number associated with the column. *row* is a one-based index of the row. *list* contains the line number to be highlighted.

change function = **"locatetab**_nnnnn_**"** (locate and activate tab item)

For a tab item in a tab group, activate the tab item. *nnnnn* contains the item number of the tab to be made active.

change function = **"mark**_nnnnn_**"** (set status to checked, on or pushed)

For a menu checkitem, make the item checked.

For a check-mark box or radio button, turn the button on.

For a toolbar lock button, lock (or push) the button.

change function = **"mark**_nnnnn_**[**_row_**]"** (mark a checkbox in a table cell)

For a table, change the selection status of the checkbox in the cell identified by column item number *nnnnn* and row number specified *row*.

change function = **"minsert**_nnnnn_**"** (insert multiple items)

For a listbox, listboxhs, mlistbox, mlistboxhs or dropbox control, *list* contains a comma delimited list of text items that will be inserted.

For a cdropbox column in a table, *nnnnn* is the item number associated with the column. *list* contains a comma delimited list of text items that will be inserted.

For all controls, if the text of one of the items in the list contains a comma, it can be embedded in a line by preceding it with a backslash. Individual lines are limited to 1024 characters.

change function = **"minsert**_nnnnn_**[**_row_**]"** (insert multiple lines in a dropbox in a table)

For a dropbox column in a table, *nnnnn* is the item number associated with the column. *row* is a one-based index of the row. *list* contains a comma delimited list of text items that will be inserted. If the text of one of the items in the list contains a comma, it can be embedded in a line by preceding it with a backslash. Individual lines are limited to 1024 characters.

change function = **"multiple"** (do multiple change functions in a single statement)

Do multiple change functions as specified by the value in *list*. The syntax of the string value of *list* is made up of one or more *change-command* values. A *change-command* value is one of these: **{** *cmd* **}** or **{** *cmd* **;** *list* **}** where *cmd* is any change function and *list* is the data associated with the *cmd*. When multiple *change-command* values are specified, there are no blanks between the braces. The backslash (****) is used before any comma or closing brace that is part of the data value in *list* associated with each *cmd*. The limit of the size of the *list* in the change multiple function is 64,000 characters.

change function = **"namefilter"** (set file name filter)

For an open file or save as file dialog, set the name filters as specified in *list*. The format of the list is a string which contains a comma separated set of values taken in pairs. The first of each pair is the name of the filter, the second is a file name mask.

change function = **"paste***nnnnn***"** (paste text)

For an edit control, if *list* is specified, it contains the text that replaces the selected text in the edit control specified by *nnnnn*. If *list* is omitted, then the selected text is deleted.

change function = **"position***nnnnn***"** (set position)

For a scroll bar and progressbar, *list* contains the value of new position of the slider. The position must be five digits long and must be blank or zero filled on the left.

change function = **"pushbutton***nnnnn***"** (make pushbutton normal)

For a pushbutton, make the pushbutton specified by *nnnnn* be a normal pushbutton.

change function = **"range***nnnnn***"** (set range)

For a scroll bar and progressbar, *list* contains a string of 15 characters with format *lllllhhhhhppppp*. *lllll* is the new low value of the range. *hhhhh* is the new high value of the range. *ppppp* is the page size of the scroll bar. These values must be five digits long and must be zero or blank filled on the left.

change function = **"readonly***nnnnn***"** (change control to the readonly state)

For any panel or dialog control, change the control to accept focus through mouse click or keyboard action, but do not allow the value of the control to be altered by the user.

change function = **"removecontrol***nnnnn***"** (remove a control from a toolbar)

For a toolbar, remove the control identified by *nnnnn.*

change function = **"removespaceafter***nnnnn***"** (remove a space after a control in a toolbar)

For a toolbar, remove the space after the control identified by *nnnnn.*

change function = **"removespacebefore***nnnnn***"** (remove a space before a control in a toolbar)

For a toolbar, remove the space before the control identified by *nnnnn.*

change function = **"rightclickoff"** (no right button messages)

This function causes right button clicked messages to be suppressed. This is the default state.

change function = **"rightclickon"** (send right button click messages)

This function causes **RBDN**, **RBUP** and **RBDC** messages to be reported for controls, including controls in table cells.

change function = **"rootlevel***nnnnn***"** (position to root level)

For a tree control, change the current position to the base level.

change function = **"select***nnnnn***"** (select item)

For a multi-line list box, *list* contains the text of an item to be highlighted.
For a tree control, the line at the current position will be selected.

change function = **"selectall***nnnnn***"** (select all items)

For a multi-line list box, all lines are highlighted.

change function = **"selectline***nnnnn***"** (select item by line number)

For a multi-line list box, *list* contains the line number of the line to be highlighted.

change function = **"setline***nnnnn***"** (set current line by number)

For a tree control, *list* contains the line number in the current indent level that will become the new current line position for the next tree control change function. If the line number is invalid, an I794 error will occur.

change function = **"setselectedline"** (set the highlighted line in a tree)

For a tree control, the line that is currently highlighted will become the new current line.

change function = **"setmaxchars***nnnnn***"** (set maximum characters)

For an ledit or mledit control, *list* contains a 5 digit value that is the new maximum number of characters that can be entered in the control specified by *nnnnn*.

change function = **"show***nnnnn***"** (show menu entry)

For a menu item, show the menu item.

change function = **"showonly***nnnnn***"** (inactive control)

For all panel or dialog controls, there is no response to mouse clicks, but the control does receive keyboard focus. This function is not applicable to trees.

change function = **"text***nnnnn***"** (set text)

For a menu item, *list* contains the new text of the item.
For a pushbutton control, *list* contains the new text for the button.
For a toolbar button, *list* contains the new tooltip string.
For an edit field, *list* contains the new value of the field.
For a vtext control, *list* contains the new text for the control.
For a checkbox control, *list* contains the new text for the control.
For a radio button control, *list* contains the new text for the control.
For a tab in a tabgroup, *list* contains the new text for the tab.

change function = **"text***nnnnn***[***row***]"** (change the text in a table cell)

For popbox, edit, vtext or rvtext cell in a table, *list* contains the new text for that control. The item number of the column is specfied by *nnnnn* and the row is specified by *row*.

change function = **"textbgcolor***nnnnn***"** (set the background color of a popbox)

For a popbox control, change the background color. The value in *list* must be a valid color.

change function = **"textbgcolordata***nnnnn***"** (set the background color of a line)

For a listbox, listboxhs, mlistbox, mlistboxhs, dropbox or tree control, change the background color of an individual line by text. The value in *list* must be a valid color, followed by a space, then the text of the line.

change function = **"textbgcolorline***nnnnn***"** (set the background color of a line)

For a listbox, listboxhs, mlistbox, mlistboxhs, or dropbox control, change the background color of an individual line by index. The value in *list* must be one a valid color, followed by a space, then the index of the line.

change function = **"textcolor***nnnnn***"** (set text color)

For a vtext, rvtext, tree, edit type control, or listbox type control not in a table, set the color of the text in the control specified by *nnnnn* to the value specified in *list*.

For a vtext or edit type control in a table, *nnnnn* is the item number of a column. Set the color of the column to the value specified in *list*.

For a table control, if *nnnnn* is the item number of a table, then set the color of all vtext and edit type controls in the table to the value specified in *list*.

The color of a specific vtext or edit type control in a table is ascertained by the priority of the specification of a cell location, not by the most recent change function to have been executed. Setting the text color of an entire table is the lowest priority. Setting the color of a column is the next higher priority. Set the color of a row is the next higher priority. Setting the color of an individual cell is the highest priority.

For all controls, the value in *list* must a valid color. If the control is a table or table column, the color **none** can also be specified which removes the table or column from the color decision process.

change function = **"textcolor***nnnnn***[***row***]"** (set text color)

For a vtext and edit type control in a table, if *nnnnn* is the item number of a table, set the color of a row in the table to the color specified by *list*. If *nnnnn* is the item number of a column of the table, set the color of an individual cell in the table to the color specified by *list*. The row index is specified by *row.* See the previous change function for how the color of a specific cell is decided. The value in *list* must be a valid color, or **none** can also be specified which removes the row or cell from the color decision process.

change function = **"textcolordata***nnnnn***"** (set the color of a line)

For a listbox, listboxhs, mlistbox, mlistboxhs, dropbox, or tree control, change the color of an individual line by text. The value in *list* must be a valid color, followed by a space, then the text of the line.

For a tree control, the index is for a line at the current indent level.

change function = **"textcolorline*nnnnn*"** (set the color of a line)
For a listbox, listboxhs, mlistbox, mlistboxhs, dropbox, or tree control, change the color of an individual line by index. The value in *list* must be a valid color, followed by a space, then the index of the line.
For a tree control, the index is for a line at the current indent level.

change function = **"textfont*nnnnn*"** (set text font)
For a vtext, rvtext and cvtext type control not in a table, set the font of the text in the control specified by *nnnnn* to the font value specified in *list*.

change function = **"textstyledata*nnnnn*"** (change the text and text style of a line)
For a listbox, listboxhs, mlistbox, mlistboxhs, tree or dropbox control, change the style of an individual line of text. The value in *list* must be one of: **bold**, **italic**, **bolditalic**, or **plain**, followed by a blank, then the text of the line. The current line of a tree is made to be the selected line.

change function = **"textstyleline*nnnnn*"** (change the text style of a line)
For a listbox, listboxhs, mlistbox, mlistboxhs, tree or dropbox control, change the style of an individual line by index. The value in list must be one of: **bold**, **italic**, **bolditalic**, or **plain**, followed by a blank, then the index of the line.

change function = **"title"** (change dialog title)
For a dialog, if *list* is specified, it contains the new title of the dialog. If *list* is omitted, then the title is removed from the dialog. This change function works only on regular dialog resources, and not for special dialog resources.

change function = **"unmark*nnnnn*"** (uncheck, turn off or unpush a control)
For a menu checkitem, make the item not checked.
For a check-mark box or radio button, turn the button off.
For a toolbar lock button, unlock the button.

change function = **"unmark*nnnnn*[*row*]"** (uncheck a cell of a table)
For a checkbox in a table, uncheck a the checkbox identified by column whose item number is *nnnnn* and whose row is specified by *row*.

change function = **"uplevel*nnnnn*"** (position up one level)
For a tree control, increase the level one away from the root level. There does not have to be anything at an indent level for this function to be valid, as long as there exists a line at one higher level, or the current position is at base level.

Messages

Windows and resources that are linked to queues generate messages as a result of user actions. These messages are sent to the DB/C program through the queue.

The get statement

The syntax of the get statement is:
get *queue* **;** *list*

Messages sent from a window may be retrieved from a queue by the get statement. The get statement moves a message from the queue into the list of variables.

Window device message formats

Queue messages from a window contain these five fields:

characters 1 - 8 window name
characters 9 - 12 message function (see be low)
characters 13 - 17 reserved
characters 18 - 22 *nnnnn* = horizontal position
characters 23 - 27 *nnnnn* = vertical position
characters 28 - 32 for mouse button click messages this is **"CTRL "** or "**SHIFT"** if those keys are depressed at the time of the click

The message function may be one of these:

POSN mouse position reported
LBDN left button was pressed
LBUP left button was released
LBDC left button was double clicked
MBDN middle button was pressed
MBUP middle button was released
MBDC middle button was double clicked
RBDN right button was pressed
RBUP right button was released
RBDC right button was double clicked
CHAR an alphanumeric key has been depressed in window without a panel, character is at position 18 of message.
CLOS close action from the title bar.
NKEY an extend keyboard key has been depressed in window without a panel, key value is stored in the horizontal position field.
WACT window notification message that window is becoming the active window
WMIN window notification message that the window is being minimized
WSIZ window has been resized. The new window width is stored in the horizontal position field and the new window height is stored in the vertical position field.
WPOS window has been moved. The position of the upper left corner with respect to the upper left corner of the main screen is stored in the horizontal and vertical position fields.
POST mouse has moved past window top
POSB mouse has moved past window bottom
POSL mouse has moved left of left side of window
POSR mouse has moved right of right side of window
HSA– horizontal scroll bar left arrow clicked
HSA+ horizontal scroll bar right arrow clicked
HSP– horizontal scroll bar left page bar clicked
HSP+ horizontal scroll bar right page bar clicked
HSTM horizontal scroll bar track movement message (button down)
HSTF horizontal scroll bar track final message (button up)
VSA– vertical scroll bar up arrow clicked
VSA+ vertical scroll bar down arrow clicked
VSP– vertical scroll bar up page bar clicked
VSP+ vertical scroll bar down page bar clicked
VSTM vertical scroll bar track movement message (button down)
VSTF vertical scroll bar track final message (button up)

Timer device message formats

This queue message is sent by a timer device whenever a timer tick occurs:

characters 1 - 8 timer name
characters 9 - 12 **TICK**
characters 13 - 17 reserved
characters 18-33 *yyyymmddhhmmsspp*

Resource message formats

In the following message formats that do not use positions 13-17, those positions will be blank filled. However, if the receiving field in those byte positions is a numeric variable, it will not be changed.

When a file open dialog, file save as dialog, font name picker dialog, or color picker dialog completes successfully (OK was pressed), this queue message is sent:

characters 1 - 8	Resource name (blank filled on right)
characters 9 - 10	**OK**
characters 11 - 12	blank
characters 13 - 17	reserved
characters 18 - end	file name, font name, color (*rrrrrgggggbbbbb*)

When an alert dialog completes (OK was pressed), this queue message is sent:

characters 1 - 8	Resource name (blank filled on right)
characters 9 - 10	**OK**

When a file open dialog, file save as dialog, font name picker dialog, or color picker dialog completes unsuccessfully (Cancel was pressed), this queue message is sent:

characters 1 - 8	Resource name (blank filled on right)
characters 9 - 12	**CANC**

When a menu item is selected, the queue message from a menu resource contains these three fields:

characters 1 - 8	Resource name (blank filled on right)
characters 9 - 12	**MENU**
characters 13 - 17	*nnnnn* is item number of menu item selected

Control box resources (dialog and panel) send several types of queue messages.

All messages from a control box resource have the same first eight bytes:

characters 1 - 8	Resource name (blank filled on right)

When the close button in the title bar of a dialog is pressed, a close message is sent. The format of this message is:

characters 1-8	dialog name
characters 9-12	**CLOS**

A panel or dialog box control receive focus message has this format:

characters 1 - 8	Resource name (blank filled on right)
characters 9 - 12	**FOCS**
characters 13 - 17	*nnnnn* For a table, *nnnnn* is item number of the column that contains the cell getting focus. For all other controls: *nnnnn* is item number of control getting focus
characters 18 - 22	*nnnnn* For a table: *nnnnn* is index (one based) of the row containing the cell getting focus

A panel or dialog box control loss of focus message has this format:

characters 1-8	dialog name
characters 9 - 12	**FOCL**
characters 13 - 17	*nnnnn* For a table: *nnnnn* is item number of the column that contains the cell losing focus For all other controls: nnnnn = item number of control losing focus
characters 18 - 22	*nnnnn* For a table: *nnnnn* is index (one based) of the row containing the cell losing focus

A push button, popbox or table columm heading pressed message has this format:

characters 1 - 8	Resource name (blank filled on right)
characters 9 - 12	**PUSH**
characters 13 - 17	*nnnnn* For a popbox cell in a table: *nnnnn* is item number of the column that contains the control For all other controls: *nnnnn* is item number of the pushbutton, popbox or table column
characters 18 - 22	*nnnnn* For a popbox cell in a table: *nnnnn* is index (one based) of the row containing the control

A general item changed message has this format:

characters 1 - 8	Resource name (blank filled on right)
characters 9 - 12	**ITEM**
characters 13 - 17	*nnnnn* is item number changed
characters 18 - end	*text* For lock buttons on a tool bar: **Y** or **N**

For radio buttons: **Y**
For check-mark boxes: **Y** or **N**
For edit and text fields: text value
For listbox, listboxhs, or dropbox controls: the text value of the selected entry, unless **lineon** is in effect, in which case this field is the line number of the selected entry
For mlistbox or mlistboxhs controls: if **lineon** is in effect, a comma separated list of selected items, or if **lineoff** is in effect, a comma separated list of one-based line numbers of the selected lines
For tab group tabs: page activated
For tree control: the text value of the selected entry, unless lineon is in effect, in which case this field is a comma delimited list of positions at each level leading up to the selected item

A general table cell changed message has this format:

characters 1 - 8	Resource name (blank filled on right)
characters 9 - 12	**ITEM**
characters 13 - 17	*nnnnn* is item number of the column
characters 18 - 22	*nnnnn* is one-based row number
characters 23 - end	text

For edit cells: text value
For checkbox cells: **Y** or **N**
For dropbox or cdropbox cells: the text value of the selected entry, unless **lineon** is in effect, in which case this field will be the five digit line number of the selected entry

NOTE: ITEM messages will not be received unless enabled by the itemon change function.

A listbox, mlistbox, or tree control double clicked message has this format:

characters 1 - 8	Resource name (blank filled on right)
characters 9 - 12	**PICK**
characters 13 - 17	*nnnnn* is list box or tree item number
characters 18 - end	text field of picked entry

A tree parent item expanded message has this format:

characters 1 - 8	Resource name (blank filled on right)
characters 9 - 12	**OPEN**
characters 13 - 17	*nnnnn* is tree control item number
characters 18 - end	text label of expanded item

A tree parent item collapsing message has this format:

characters 1 - 8	Resource name (blank filled on right)
characters 9 - 12	**CLOS**
characters 13 - 17	*nnnnn* is tree control item number
characters 18 - end	text label of collapsed item

A right button message has this format:

characters 1 - 8	Resource name (blank filled on right)
characters 9 - 12	**RB***xx* is right button message where *xx* is **DN** for right button down *xx* is **UP** for right button up *xx* is **DC** for right button double clicked
characters 13 - 17	*nnnnn* is control item number
characters 18 - 22	*nnnnn* is horizontal position relative to the window
characters 23 - 27	*nnnnn* is vertical position relative to the window
characters 28 - 32	**CTRL** or **SHIFT** if those keys are depressed at the time of the click

A scroll bar positioning message has this format:

characters 1 - 8	Resource name (blank filled on right)
characters 9 - 12	**SB***xx* is scroll bar message where
	xx is **A−** for scroll bar up or left arrow clicked
	xx is **A+** for scroll bar down or right arrow clicked
	xx is **P−** for scroll bar page up or page left clicked
	xx is **P+** for scroll bar page down or page right clicked
	xx is **TM** for track movement message (left button pressed)
	xx is **TF** for track final message (left button up)
characters 13 - 17	*nnnnn* is scroll bar item number
characters 18 - 22	*nnnnn* is new current position

An edit control text selected message is produced whenever the selected text of a control changes. The format of this message is:

characters 1 - 8	Resource name (blank filled on right)
characters 9-12	**ESEL**
characters 13-17	*nnnnn* is edit control item number
characters 18 - end	selected text, if null then no text is selected

When a panel or dialog has keyboard focus, and one of the function keys **F1** through **F12** is pressed, and there is no corresponding keyboard accelerator key defined in the current menu, then an **NKEY** message is produced. When there is no control with focus, the **NKEY** message is sent with the window name as the resource name. The **NKEY** message is also produced when the **Esc** key is pressed and there is no escpushbutton defined in the window. The format of this message is:

characters 1 - 8	Resource name (blank filled on right)
characters 9-12	**NKEY**
characters 13-17	*nnnnn* is control item number
characters 18-22	*nnnnn* is key value (**F1** = 301 ... **F12** = 312, **ESC** = 257)

Application Device

The application device provides a method for closing the console window, and a method for changing the taskbar text and icon of the main DB/C DX runtime window. The application device is accessed by opening a device variable with the name **application**. After the application device has been opened, the change functions can be used on the device.

The syntax of the change statement is:

change *device* **,** *charexp* **;** *list*

where the logical string of *charexp* is the change function.

These change functions are available:

change function = **"consolewindow"**
The console window will be closed if the value of *list* is **off**.

change function = **"consolefocus"**
The console window will be given focus.

change function = **"dialogicon"**
When using **dbc.exe** in Windows, an icon will appear in the upper left corner of dialogs. *list* contains the name of a valid 16 by 16 or 32 by 32 icon resource.

change function = **"desktopicon"**
When using **dbc.exe** in Windows, an icon will appear in the taskbar for the DB/C DX runtime if *list* contains the name of a valid 16 by 16 or 32 by 32 icon resource. If the icon name specified in *list* is a 32 by 32 icon, then it will be the large icon seen in the **ALT+TAB** window for the DB/C DX runtime.

change function = **"taskbartext"**

When using **dbc.exe** in Windows, the text for the DB/C DX runtime window in the taskbar will change to the text specified in *list*.

Clipboard Device

The clipboard is a device that text and images may be copied to and from. The clipboard may be accessed by opening a device variable with the name **clipboard** and using the load and store statements.

Here is an example:

```
clipbd device
char100 char 100
         open clipbd, "clipboard"
         load char100 from clipbd
         store "new text on clipboard" to clipbd
         close clipbd
```

The example first retrieves the first 100 characters of the text that is already in the clipboard. The text new text on clipboard is then moved to the clipboard, destroying the text information previously stored there.

Here is an example that retrieves and displays an image stored on the clipboard:

```
pic image h=600, v=400
win device
clp device
         prepare win, "window=window01, size=600:400"
         open clp, "clipboard"
         load pic from clp
         close clp
         show pic in win
```

The query statement may be used to retrieve the size of the image stored in the clipboard.

Fonts

The draw and print statements allow the text font to be set. For draw, the syntax is *font=charvar* or **font=***charlit*. For print, the syntax is ***font=***charvar* or ***font=***charlit*. The syntax of the logical string of *charvar* or *charlit* is:

font **(***attribute* **,** ...**)**

These five fonts are always available: **Courier**, **Helvetica**, **Monaco**, **System**, and **Times**. The fonts **Symbol** and **Zapfdingbats** are available when printing to a PDF document. Other fonts may be available—see the chapter about your particular operating system for more information. The font names and attributes may be in upper-case, lower-case, or in mixed upper/lower-case.

These *attribute*s are always available: **bold**, **nobold**, **italic**, **noitalic**, **underline**, **nounderline**, and **plain**. **plain** is equivalent to the **nobold**, **noitalic**, **nounderline combination**. If a number between 1 and 72 is specified as an *attribute*, it is the font size (in points).

Only those attributes and font names specified are changed. For example:

```
         print *font="times(12,plain)", "A"
         print *font="(italic)", "B"
         print *font="courier(14)", "C"
```

In this example, **A** will be printed in regular Times 12 point. **B** will be printed in Times 12 point italic. **C** will be printed in Courier 14 point italic.

TIFF, GIF, PNG, and JPEG File Support

The load and store statements, in conjunction with device and image variables, allow for access to or creation of Tagged Image File Format files (TIFF files), access to Graphics Interchange Format files (GIF files), access to Portable network Graphics files (PNG files), and access to Joint Photographic Experts Group files (JPEG files). The load statement is used to copy the image from the TIFF, GIF, PNG or JPEG file to an image variable. The store statement copies an image from an image variable into a TIFF file.

When an open or prep statement is executed on a device variable and the filename extension is **.tif**, then the file is assumed to be a TIFF file. When an open statement is executed on a device variable and the filename extension is **.gif**, then the file is assumed to be a GIF file. When an open statement is executed on a device variable and the filename extension is **.png**, then the file is assumed to be a PNG file. When an open statement is executed on a device variable and the filename extension is **.jpg**, then the file is assumed to be a JPEG file.

The open statement also supports TIFF, GIF, PNG and JPEG files that have extensions other than **.tif**, **.gif**, **.png** and **.jpg**. This is done by appeding **(tif)**, **(gif)**, **(png)** or **(jpg)** after the file name. For example **"picture.001(tif)"** specifies that the file is a TIFF file.

The open statement also supports multi-image TIFF files. The image to load is specified as a numeric index enclosed by parentheses and following the **.tif** extension in the filename. For example, **"picture.tif(2)"** is the second image in the TIFF file and **"picture.001(2,tif)"** is also the second image in the TIFF file.

The query statement can be used to retrieve information about an image device.

The query function **"imagesize"** is used to obtain the horizontal and vertical size of the image file. The return value is ten characters in the form *hhhhhvvvvv*.

The query function **"bitsperpixel"** retrieves the number of bits contained in each pixel in the image. The return value is five characters in the form *nnnnn*.

The query function **"imagecount"** retrieves the number of images stored in the TIFF file. The return value is five characters in the form *nnnnn*.

The query function **"resolution"** returns a value, in the form *hhhhhvvvvv*, which represents the horizontal and vertical image resolution respectively in pixels per inch. If the image file does not contain resolution information, or the image type is gif, the horizontal and vertical resolution values will be set to zero.

Here is an example:

```
win device
tiffile device
img image @
hv dim 10
h dim 5
v dim 5
bpp dim 5
         prepare win, "window=w01"
         open tiffile, "picture.tif"
         query tiffile, "imagesize"; hv
         unpack hv to h, v
         query tiffile, "bitsperpixel"; bpp
         makevar ("I" + h + "," + v + "," + bpp) to img
         load img from tiffile
         show img on win
```

Error Codes and Messages

Class	Trap	Code	Meaning
C	CFAIL	101	Program module not found during chain, loadmod, ploadmod or make
C	CFAIL	102	Invalid program
C	CFAIL	103	Program is too large
C	CFAIL	104	Common data area does not align
C	CFAIL	105	Access violation during chain
C	CFAIL	106	Null or invalid file name
C	CFAIL	107	Out of memory during chain, loadmod, or make
C	CFAIL	153	Attempt to loadmod existing preloaded or chain module
C	CFAIL	154	Program module previously loaded with different time stamp during chain, loadmod, ploadmod or make
C	CFAIL	155	Global type does not match previous definition
C	CFAIL	157	Class not found in program module during make
C	CFAIL	158	Inherited class variable not found during make
C	CFAIL	159	Inherited class variable does not match ancestor's definition during make
C	CFAIL	161	Unable to connect to file server
C	CFAIL	162	Error communicating with file server
F	FORMAT	201	Attempt to read non-numeric data
R	RANGE	301	Attempt to read past file end
R	RANGE	302	Array index reference out of bounds
P	SPOOL	401	Unable to create print file or open device
P	SPOOL	402	Device or print file name is null or invalid
P	SPOOL	403	Print device is off-line or otherwise unavailable
P	SPOOL	404	Error during print
P	SPOOL	405	Access violation during
P	SPOOL	406	Operation attempted on a closed print file
P	SPOOL	407	Invalid tab value used with print
P	SPOOL	408	Out of memory during print operation
P	SPOOL	409	Open mode conflict during splopen
P	SPOOL	410	Unable to extend file during print
P	SPOOL	411	Invalid option used with splopen
P	SPOOL	452	Invalid option used with splopen
E		501	Return stack overflow during call or perform
E		502	Return stack overflow during call, perform, make, or destroy

Class	Trap	Code	Meaning
E		503	Return stack empty
E		504	Another filepi is already active
E		505	Execution error (invalid instruction or control code)
E		506	Invalid data used with scrnrestore, staterestore, winrestore or traprestore
E		507	Array index reference out of bounds
E		551	External label or method not found
E		552	Address variable contains invalid address
E		555	getparm or loadparm with invalid data pointer
E		556	Out of memory during makevar
E		557	Attempt to modify a literal value, non-numeric variable, or non-character variable
E		558	return or trap attempted to an unloaded module
E		559	Exceeded 10 nested lists or array variables
E		560	Invalid use of a label variable
E		561	Conflict with internal pfile name
E		562	Class make routine or destroy routine not found
E		563	Call to method with uninstantiated object
E		564	Attempt to use an image variable in a non-GUI runtime
E		566	A truncation occurred during evaluation of a character expression
I	IO	601	File not found during open, rename or erase
I	IO	602	Open mode conflicts with another program
I	IO	603	Null or invalid file name used with open, prepare, rename or erase
I	IO	604	Invalid file type used with open, prepare, rename or erase
I	IO	605	Attempt to prepare or rename a file that already exists
I	IO	606	Key length specified in open differs from index file
I	IO	607	Access violation during open, prepare, rename or erase
I	IO	608	Associative key specification in prepare missing or invalid
I	IO	609	Index key specification invalid in prepare
I	IO	610	Invalid record length specified in open or prepare
I	IO	611	Attempt to rename a file to a different resource
I	IO	612	Error during close
I	IO	613	Unable to create file during prepare
I	IO	614	Out of memory during open, prepare, rename or erase
I	IO	651	Native file open error
I	IO	653	Unable to find dynamic link library

Class	Trap	Code	Meaning
I	IO	654	Dynamic link library incompatible or invalid
I	IO	655	Error loading user routine from dynamic link library
I	IO	656	Error when an ifile open is attempted for a file that allows duplicate keys and the ifile statement contains the nodup keyword
I	IO	657	Unable to connect to file server
I	IO	658	Error communicating with file server
I	IO	701	File not open - operation attempted on closed file
I	IO	702	Attempt to write to read-only file
I	IO	703	Attempt to randomly access a variable length record file
I	IO	704	Invalid tab value used with read, updatab or writab
I	IO	705	Attempt to access a file that is off-line or otherwise not available
I	IO	706	Error during write
I	IO	707	Access violation during read, write or writab
I	IO	708	Attempt to write or insert with null index key
I	IO	709	Attempt to write or insert a duplicate key
I	IO	710	Attempt to update to invalid file position
I	IO	711	Attempt to update a compressed file
I	IO	712	Associative read key specification error
I	IO	713	Associative read insufficient key(s) specified
I	IO	714	Invalid associative index position for readkg or readkgp
I	IO	715	Attempt to read a record that is too small
I	IO	716	Attempt to read, write or update a record that is too large
I	IO	717	Invalid data file position for insert or delete
I	IO	718	Attempt to perform an indexed read with null key and invalid position
I	IO	719	Attempt to insert two identical keys for the same record
I	IO	720	Attempt to read a record that does not exist or was deleted
I	IO	721	Specified key too long in read, write, insert, delete or deletek
I	IO	751	comfile not open
I	IO	752	SQL error
I	IO	753	Communications error
I	IO	754	Communications not available
I	IO	755	Invalid queue handle
I	IO	761	Error creating device—invalid device name
I	IO	762	Error opening device—invalid device name
I	IO	763	Error creating resource—invalid resource type
I	IO	764	Error opening resource—invalid resource type

Class	Trap	Code	Meaning
I	IO	765	Error creating device—system error
I	IO	766	Error opening device—system error
I	IO	767	Error creating resource—system error
I	IO	768	Error opening resource—system error
I	IO	771	Error creating device or resource—syntax error
I	IO	772	Error creating device or resource—expecting comma
I	IO	773	Error creating device or resource—expecting equal sign
I	IO	774	Error creating device or resource—expecting colon
I	IO	775	Error creating device or resource—expecting value
I	IO	776	Error creating device or resource—expecting numeric value
I	IO	777	Error creating device or resource—invalid use of forcing character
I	IO	778	Error creating device or resource—invalid font name
I	IO	779	Error when an icon resource name that is specified in a prepare statement of a resource does not exist
I	IO	780	Unable to connect to file server
I	IO	781	Error communicating with file server
I	IO	791	Device or resource not open
I	IO	792	Invalid device or resource type
I	IO	793	Error during query—invalid query function or value
I	IO	794	Error during change—invalid change function or value
I	IO	798	Error communicating with Smart Client
I	IO	799	Unspecified error
E		1155	Exceed maximum files open
C	CFAIL	1167	Unable to allocate memory
C	CFAIL	1168	Unable to read
C	CFAIL	1180	Unable to seek
C	CFAIL	1181	Bad library
C	CFAIL	1182	Invalid value for file handle
C	CFAIL	1184	Unable to open
C	CFAIL	1185	Invalid argument
C	CFAIL	1186	No semaphores
C	CFAIL	1199	Unspecified error
P	SPOOL	1451	File not open
P	SPOOL	1455	Exceed maximum files open

Class	Trap	Code	Meaning
P	SPOOL	1457	Invalid file type
P	SPOOL	1467	Unable to allocate memory
P	SPOOL	1468	Unable to read
P	SPOOL	1469	Unable to write
P	SPOOL	1480	Unable to seek
P	SPOOL	1482	Invalid value for file handle
P	SPOOL	1484	Unable to open
P	SPOOL	1486	No semaphores
P	SPOOL	1487	Error opening or reading print map
P	SPOOL	1498	Exceeded user license
P	SPOOL	1499	Unspecified error
I	IO	1655	Exceed maximum files open
I	IO	1663	Invalid index file
I	IO	1664	Wrong key length
I	IO	1665	Wrong record length
I	IO	1667	Unable to allocate memory
I	IO	1668	Unable to read
I	IO	1669	Unable to write
I	IO	1670	Unable to delete
I	IO	1671	Unable to lock file
I	IO	1678	Unable to rename
I	IO	1680	Unable to seek
I	IO	1681	Bad library
I	IO	1682	Invalid value for file handle
I	IO	1684	Unable to open
I	IO	1686	No semaphores
I	IO	1688	Error opening or reading collate file
I	IO	1689	Error opening or reading casemap file
I	IO	1698	Internal programming error
I	IO	1699	Unspecified error
I	IO	1751	File not open
I	IO	1752	File not available
I	IO	1753	Access denied
I	IO	1755	Exceed maximum files open

Class	Trap	Code	Meaning
I	IO	1757	File type does not support corresponding IO operation
I	IO	1758	No end-of-record mark or record too long
I	IO	1759	Record too short
I	IO	1760	Invalid character encountered
I	IO	1761	Beyond end-of-file
I	IO	1762	Record has already been deleted
I	IO	1763	Index file is invalid
I	IO	1766	Invalid key
I	IO	1767	Unable to allocate memory
I	IO	1768	Unable to read
I	IO	1769	Unable to write
I	IO	1771	Unable to lock file or record
I	IO	1780	Unable to seek
I	IO	1781	Bad library
I	IO	1782	Invalid value for file handle
I	IO	1783	Attempt write on read-only file
I	IO	1784	Unable to open
I	IO	1786	No semaphores
I	IO	1787	No EOF
I	IO	1798	Internal programming error
I	IO	1799	Unspecified error