Reference Manual

ACUCOBOL-GT®

Version 8.1.3

Micro Focus
9920 Pacific Heights Blvd.
San Diego, CA 92121
858.790.1900

E-01-RM-100501-ACUCOBOL-GT-8.1.3

Contents

Chapter 1: Introduction

Chapter 2: Program Structure

Chapter 3: Identification Division

Chapter 4: Environment Division

Chapter 5: Data Division

Chapter 6: Procedure Division

1 Introduction

Key Topics

1.1 Overview of Reference Manual

ACUCOBOL-GT® is part of the ***extend***® family of Micro Focus solutions. ACUCOBOL-GT is an implementation of the COBOL-1985 programming language (ANSI X3.23-1985 and the ANSI X3.23a-1989 supplement). This manual provides a full technical description of the ACUCOBOL-GT language. It is written in a style similar to the official ANSI definition of COBOL and is designed for experienced COBOL programmers. Intended as a reference, it does not try to teach programming or the COBOL language. Readers interested in a more general introduction to COBOL may want to consider classes or commercially available textbooks.

Note: Software installation instructions are located in a separate *Getting Started* guide.

1.2 Conventions

This manual uses a meta-language to describe the ACUCOBOL-GT syntax. This meta-language follows the conventions used in most COBOL manuals as well as the ANSI standard. These conventions are described below.

Unless otherwise indicated, the references to "Windows" in this manual denote the following 32-bit versions of the Windows operating systems: Windows Vista, Windows XP, Windows NT 4.0 or later, Windows 2000, Windows 2003; and the following 64-bit versions of the Windows operating system: Windows Server 2003 and 2008 x64, Vista x64. In those instances where it is necessary to make a distinction among the individual versions of those operating systems, we refer to them by their specific version numbers ("Windows 2000," "Windows NT 4.0," etc.).

1.2.1 Upper-case and Special-character Words

Underlined upper-case words are keywords. A keyword is required when it is encountered in the syntax. The following special characters are not underlined but are required when they appear: +, -, <, >, *, /, **, <=, >=, =, colon and period.

Upper-case words that are not underlined are optional. They serve only to improve the readability of the source program.

1.2.2 Lower-case Words

Lower-case words serve as generic items. They can indicate COBOL variables, literals, PICTURE elements, or other syntactical elements. When referred to in the text, a generic item is shown in *italics*.

1.2.3 Brackets, Braces and Vertical Bars

Brackets ([]) enclose optional elements. When several bracketed entries are stacked vertically, then you can select one (but not more than one) of these entries.

Braces ({}) indicate that you *must* select one (but not more than one) of the enclosed vertically stacked entries. If only one entry appears, then the braces serve as delimiters for repetition (see Ellipses below).

1.2.4 Vertically stacked entries enclosed in vertical bars indicate that you may select one or more of the entries. Any number of entries can be selected, but no entry may be selected more than once. Ellipses

Ellipses (. . .) indicate repetition. The immediately preceding element may be repeated any number of times. If an element consists of a required phrase, the phrase is enclosed in braces.

1.2.5 Shading

Shaded areas are not currently used in the syntax.

1.3 Acknowledgment

Much of the material in this manual is from the ANSI X3.23-1985 COBOL standard and the ANSI X3.23a-1989 supplement. The following statement is required by the standards document.

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the CODASYL COBOL Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

The authors and copyright holders of the copyrighted material used herein are: FLOW-MATIC (trademark of Sperry Rand Corporation), Programming for the UNIVAC (R) I and II, Data Automation Systems copyrighted 1958, 1959 by Sperry Rand Corporation; IBM Commercial Translator Form No. F28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell.

They have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorizations extend to the reproduction and use of COBOL specifications in programming manuals or similar publications.

2 Program Structure

Key Topics

2.1 Language Elements

This chapter describes the basic language elements that make up a COBOL program. These elements are described fully in the following sections.

2.1.1 COBOL Words

A COBOL word is a string of not more than 60 characters, which forms a user-defined word or a reserved word. Each character of a COBOL word is selected from the set of letters, digits, underscores, and hyphens. Hyphens or underscores may not appear as the first or last character. When used in COBOL words, lower-case letters are treated as if they were upper-case, and underscores are treated as if they were hyphens.

2.1.1.1 User-defined words

A user-defined word is a COBOL word that is created by the programmer. Except for section-names, paragraph-names, segment-numbers, and level-numbers, all user-defined words must contain at least one alphabetic character. User-defined words may not be any of the reserved words. See Appendix B of Book 4 for a complete list of reserved words.

All user-defined words must be unique except as specified in the rules for uniqueness of reference. However, segment-numbers and level-numbers need not be unique.

2.1.2 Literals

A literal is a character string that defines a value. There are two types of literals: numeric and nonnumeric.

2.1.2.1 Numeric literals

A numeric literal is a character string selected from the digits, the plus sign, the minus sign, and the decimal point. Numeric literals may contain up to 18 digits. [This increases to 31 digits if 31-digit support (-Dd31) is in effect.] The following rules govern the formation of numeric literals.

1. A literal must contain at least one digit.
2. It must contain no more than one sign character and, if one is used, it must be the leftmost character of the string.
3. A literal must not contain more than one decimal point. The decimal point is treated as an assumed decimal point and may appear anywhere within the literal except as the rightmost character.

If a literal conforms to the rules for formation of a numeric literal, but is enclosed in quotation marks, it is a nonnumeric literal.

Numeric literals may also be specified using binary, octal, or hexadecimal notation. To specify a numeric literal in one of these forms, preface the number with one of the following prefixes:

Binary	"B#"
Octal	"O#" "%" is accepted in HP COBOL compatibility mode ("-Cp")
Hexadecimal	"X#" or "H#"

For example:

Number	**Binary**	**Octal**	**Hexadecimal**
3	B#11	O#3	X#3 or H#3
8	B#1000	O#10	X#8 or H#8
12	B#1100	O#14	X#C or H#C
128	B#10000000	O#200	X#80 or H#80
255	B#11111111	O#377	X#FF or H#FF

Leading zeroes after the "#" are ignored. For example, X#00FF and X#FF are equivalent.

The compiler converts each numeric literal specified in this way to an unsigned long integer. In most cases, this is a 32-bit unsigned number, so the maximum value of a numeric literal that can specified with this notation is 4294967295, or (2**32) -1.

"LENGTH OF" expression

The "LENGTH OF" expression can be used anywhere you would use a numeric literal, except as a subscript or reference modifier. The compiler treats this expression as if you have coded a numeric literal. The "LENGTH OF" expression is written as follows:

```
LENGTH OF data-name
```

Data-name can be a numeric or nonnumeric literal or the name of a data item of any type. *Data-name* may include subscripts if it refers to a table item. The compiler calculates the value of the "LENGTH OF" expression and replaces it with a numeric literal equivalent to the current number of bytes of storage used by the data item or literal referenced in "LENGTH OF." For example:

```
77 my-item PIC x(10).
78 my-item-length value LENGTH OF my-item.
```

becomes:

```
77 my-item PIC x(10).
78 my-item-length value 10.
```

The LENGTH OF expression can also be used in the procedure division as demonstrated in this example:

```
01  my-data.
    03  my-table occurs 20 times.
        05  my-element-1   pic x(10).
05  my-element-2   pic 99.

MOVE LENGTH OF my-element-1 TO data-size.
MOVE LENGTH OF my-table TO data-size.
MOVE LENGTH OF my-table(1) TO data-size.
```

In this example the compiler treats the first MOVE as MOVE 10 TO data-size, the second MOVE as MOVE 240 TO data-size, and the third MOVE as MOVE 12 TO data-size.

Note: This expression (when used on a table) works differently in ACUCOBOL-GT than in other COBOL compilers, such as IBM Enterprise COBOL. ACUCOBOL-GT returns the size of the entire table, while IBM returns the size of a single element of the table. You can use the IBM method by compiling the program with "-Cv", which turns on the compiler's IBM compatibility mode. Refer to ACUCOBOL-GT User's Guide, **Section 2.2.5** for details on the -Cv compiler option.

Floating-Point Literals

1. A floating-point literal has the following format:

```
[  +  ]  k.m  {  E  }  [  +  ]  n
[  -  ]       {  e  }  [  -  ]
```

   In the above:

   - "k.m" represents a number with at least one digit.
   - "n" represents one or more digits.
   - If the functions of the decimal point and comma are switched with DECIMAL IS COMMA, then "k.m" will be "k,m".

   Here are a few examples of floating-point numbers:

```
-12.345e12
.0123E-6
123.E1
```

2. Floating-point literals in the Procedure Division are stored internally as USAGE DOUBLE.
3. The legal range of floating-point values is determined by the target machine. If you express a literal that is out of range for a particular machine, the runtime reports a warning message and substitutes the closest boundary value--either zero or the maximum floating-point value for the machine.
4. On some computers, floating-point computations may give imprecise results. This is a hardware limitation; some floating point numbers cannot be precisely represented on some machines.

2.1.2.2 Nonnumeric literals

A nonnumeric literal (sometimes called an alphanumeric literal) is a character string delimited at the beginning and at the end by quotation marks or apostrophes. The beginning and ending delimiters must be the same (that is, either both quotes or both apostrophes).

Nonnumeric literals may be up to 4096 characters in length. The characters contained in the delimiters may be selected from all characters available on the host computer.

To place the delimiter character in a nonnumeric literal, use two contiguous delimiter characters (either two quotes or two apostrophes). These two characters represent a single occurrence of that character.

You can also specify nonnumeric literals by supplying the hexadecimal value of the characters desired using the native character set. This can be used, for example, to encode device control codes. Any of the following formats are recognized:

```
X"hex-values"
X'hex-values'
H"hex-values"
H'hex-values'
```

The initial “H” or “X” may be either upper- or lower-case. The *hex-values* consist of one or more hexadecimal digits. These digits are drawn from the set of characters ‘0’ - ‘9’ and ‘A’ - ‘F’. Every two hexadecimal digits represent one character position, with the first digit encoding the high-order 4 bits of the character, and the second digit encoding the low-order 4 bits. If an odd number of hexadecimal digits is specified, then the low-order 4 bits of the last character are treated as zeros.

Example: the following pairs of nonnumeric literals are equivalent (when the native character set is ASCII):

X”414243”	“ABC”
h’32313’	“210”
H”6E”	“n”
x”22”	“““

2.1.2.3 Figurative constants

Figurative constants are literals that are generated by the compiler through the use of reserved words. These words are described below. The singular and plural forms of the words are equivalent and may be used interchangeably.

1. **ZERO, ZEROS, ZEROES** Represents the numeric value “zero” or one or more occurrences of the character 0, depending on whether the constant is treated as a numeric or nonnumeric literal.
2. **SPACE, SPACES** Represents one or more space characters.
3. **HIGH-VALUE, HIGH-VALUES** Represents one or more characters with the highest ordinal position in the program collating sequence. Usually this is the hexadecimal value “FF”.
4. **LOW-VALUE, LOW-VALUES** Represents one or more characters with the lowest ordinal position in the program collating sequence. Usually this is the binary value 0.
5. **QUOTE, QUOTES** Represents one or more quotation mark characters. These words may not be used in place of quotation marks for delimiting nonnumeric literals.
6. **ALL *Literal*** Represents all or part of the string generated by successive concatenations of the characters comprising the *literal*. The *literal* must be nonnumeric.
7. **Symbolic Character** Represents one or more of the characters defined as the value of this symbolic character in the SYMBOLIC CHARACTERS clause of the SPECIAL-NAMES paragraph.
8. **NULL, NULLS** Represents the numeric value “zero” or one or more occurrences of a character whose underlying representation is binary zero. This also represents an invalid memory address when it is used in conjunction with POINTER data types.

The word “ALL” may be placed in front of any of the preceding forms (except the ALL literal. Its use is redundant in this case.)

When a figurative constant represents a string of one or more characters, the length of the string is determined by the compiler from the context according to the following rules.

1. When a figurative constant is specified in a VALUE clause, or when it is associated with another data item (for example, when it is moved or compared to another data item), the string of characters specified by the figurative constant is repeated character by character on the right until the size of the resultant string is equal to the number of character positions in the associated data item.

2. When a figurative constant is not associated with another data item (for example in a DISPLAY, STRING, or UNSTRING statement), the length of the string is one occurrence of the ALL *literal* or one character in all other cases.

A figurative constant is valid anywhere a literal is. However, ZERO and NULL are the only valid figurative constants for a literal restricted to numeric literals.

2.1.3 Picture Strings

A PICTURE character-string defines the size and category of an elementary data item. A PICTURE character-string consists of certain symbols, which are composed of the currency symbol and certain other characters in the COBOL character set. A full description of the format of a PICTURE string is given when the PICTURE clause is described.

Any punctuation character that appears as part of a PICTURE string is not considered a punctuation character, but rather as a symbol used in the specification of that PICTURE string.

2.1.4 Separators

A separator is a character or two contiguous characters formed according to the following rules.

1. The space character is a separator. Anywhere a space is used as a separator, more than one space may be used.

2. The comma and semicolon characters, immediately followed by a space, are separators that may be used anywhere the separator space is used. They can be used to improve program readability.

3. The period character, followed by a space, is a separator. It must be used only to indicate the end of a sentence or where required by the ACUCOBOL-GT syntax.

4. The left and right parentheses are separators. They must be used in balanced pairs.

5. The quotation mark character is a separator. An opening quotation mark must be immediately preceded by a space or left parenthesis; a closing quotation mark must be immediately followed by one of the separators space, comma, semicolon, period, or right parenthesis. Apostrophes may be substituted for quotation marks in balanced pairs.

6. The colon character is a separator that may be used only when required by the ACUCOBOL-GT syntax.

7. Spaces may immediately precede or follow any separator unless they would be enclosed by matching quotation marks. In this case, the spaces would be treated as part of the nonnumeric literal.

2.1.5 Comment Entries

A comment entry is an entry in the Identification Division that may be any combination of characters from the computer's character set. A comment-entry ends with the first line that contains text in Area A.

2.2 Source Format

The ACUCOBOL-GT compiler recognizes two source program formats: ANSI and terminal. The ANSI format conforms to the standard COBOL source format. The "terminal" format is designed for ease-of-use when you are programming from an interactive terminal. It is upward compatible with VAX COBOL terminal format.

The selection of which format to use is made at compile time. For details, see Book 1, section 2.4, “Source Formats.” The two formats are described in the following sections.

2.2.1 ANSI Format

The ANSI source format divides an input line into several fields. These are determined by character position. Each input line must be 80 characters. Input lines that are shorter than 80 characters are padded with spaces to make 80 characters, while lines longer than 80 characters are truncated on the right. Tab characters are converted into spaces such that the “tab stops” are eight characters apart.

The ANSI format has five fields. These are:

Sequence Number Area (columns 1 - 6) This area is ignored by the compiler and may contain any characters. It is traditionally used for sequence numbers to re-order a scrambled card deck.

Indicator Area (column 7) This column must contain one of the following characters:

Space	Default. The compiler processes the line normally.
Hyphen	Continuation. The compiler processes the line as a continuation of the previous line.
Asterisk	Comment. The compiler ignores the contents of the line.
Dollar sign	Comment. The compiler ignores the contents of the line.
Slash	New page. Same as asterisk, except that in the source listing created by the compiler this line starts on a new page.
“D”	Conditional debugging line. The compiler treats this line as a comment line unless the compiler is run with the option to include debugging lines, in which case the line is treated normally.

Area A (columns 8 - 11) Area A contains division headers, section headers, paragraph names, and some level indicators.

Area B (columns 12 - 72) Area B contains all other COBOL text.

Identification Area (columns 73 - 80) Any desired text may be placed here. However, the compiler can conditionally compile lines based on patterns found in this area. For details, see Book 1, **section 2.7, "Source Code Control."**

2.2.2 Terminal Format

Terminal format is convenient for interactive programming. Lines may be longer or shorter than 80 characters. Tab characters are expanded to every eight spaces. The terminal format divides the source line into four fields as follows:

Indicator Area (column 1) The contents of this area are identical to the contents of the ANSI format area of the same name, with two exceptions. If the conditional debugging indicator "D" is used, it must be preceded by a backslash (\). This places the "D" in column 2. If a normal COBOL line is desired, then the indicator area is eliminated (a space is *not* used).

Area A Starts immediately after the indicator area (either column 1, 2, or 3). It extends for 4 characters. For a standard source line, Area A starts in column 1.

Area B Starts after Area A, in column 5 or later, and extends to the end of the line or the start of the Identification Area.

Identification Area Starts when "|" or "*>" is encountered, provided it is not part of a literal. The Identification Area extends to the end of the line. This can be used to introduce in-line comments.

ACUCOBOL-GT allows up to 320 characters per line. If a line goes over the limit, the compiler issues a warning and truncates the line to 320 characters. If truncation causes an error, the compiler reports the error.

The following sample COBOL text is in terminal format:

```
* The following paragraph is a sample of terminal
* format. Notice how comments and Area A both start
* in column 1

TEST-PARAGRAPH.
      MOVE SAMPLE-1-VALUE TO SAMPLE-1.
\D    DISPLAY "SAMPLE-1 = ", SAMPLE-1.
      PERFORM EDIT-SAMPLE.
```

2.2.3 Line Continuation

Sentences, entries, phrases, and clauses that continue in Area B of subsequent lines are called continuation lines. A hyphen in a line's indicator area causes the first nonblank character in Area B to be the immediate successor of the last nonblank character of the preceding line. This continuation excludes intervening comment lines and blank lines.

If the continued line ends with a nonnumeric literal without a closing quotation mark, the first nonblank character in Area B of the continuation line must be a quotation mark. The continuation starts immediately after the quotation mark. All spaces at the end of the continued line are part of the literal.

If the indicator area of the continuation line is blank, then the compiler treats the last nonblank character of the preceding line as if it were followed by a space.

2.2.4 Blank Lines and Comment Lines

A blank line is one that contains only spaces in Area A and Area B. A comment line is one that contains an asterisk, dollar sign, or slash character in the indicator area. Conditional debugging lines are also considered comment lines unless the program is compiled with the conditional debugging option. Blank lines and comment lines may appear anywhere in the source program and have no effect. They will appear in the source listing. If a comment line contains a slash in the indicator area, then the source listing will contain a page eject prior to that line.

2.3 Compiler Compatibility Modes

The ACUCOBOL-GT compiler accepts certain variants of the COBOL language. These variants are designed to make it easier to compile programs written using other COBOL compilers. The variants, or alternate compatibility modes, are based on the following popular compilers: VAX COBOL version 4.0, Ryan McFarland COBOL version 2.X, ICOBOL, and HP COBOL II/XL. Limited compatibility with IBM DOS/VS COBOL is also available.

ACUCOBOL-GT documentation refers to these modes respectively as VAX COBOL compatibility mode, RM/COBOL compatibility mode, ICOBOL compatibility mode, HP COBOL compatibility mode, and IBM DOS/VS COBOL compatibility mode.

The compatibility mode to use for a program is selected when that program is compiled. For more information, see Book 1, **section 2.2.5, "Compatibility Options."** Different programs may use different compatibility modes, even if they are part of the same run unit.

Differences in these modes are detailed in the appropriate sections in this manual. Their differences primarily lie in the handling of files and the ACCEPT and DISPLAY verbs. Unless otherwise noted, any comments applying to VAX COBOL compatibility mode also apply to ICOBOL compatibility mode.

2.3.1 ANSI ACCEPT and DISPLAY Verbs

The ACCEPT and DISPLAY verbs have ANSI formats that provide strict compatibility with the ANSI definition of these verbs. ACCEPT and DISPLAY also have expanded ACUCOBOL-GT formats that are not ANSI-compliant but offer more functionality. The ANSI definition is quite limited in that it does not provide any screen control facilities. However, these verbs can be useful when you are:

- converting programs from other COBOL compilers
- directing messages to the runtime's error file

- providing low-level control of the user's console

The ANSI formats of ACCEPT and DISPLAY are subsets of the extended ACUCOBOL-GT formats. Thus, the compiler needs guidance in determining which format is desired. This is important because different formats result in different behavior. When examining an ACCEPT or DISPLAY statement, the compiler applies the following rules, in order:

1. If you specify FROM CRT or UPON CRT, the compiler uses ACUCOBOL-GT format.
2. If you specify a FROM or UPON phrase for a device other than CRT, the compiler uses ANSI format.
3. If you use any ACUCOBOL-GT extensions, the compiler uses ACUCOBOL-GT format.
4. If you place the phrase "CONSOLE IS CRT" in Special-Names, the compiler uses ACUCOBOL-GT format.
5. If you use the compile-time option "-Ca" (ANSI compatibility), the compiler uses ANSI format.
6. Otherwise, the compiler uses ACUCOBOL-GT format.

Note: By default, the compiler uses ACUCOBOL-GT format.

2.4 Source Management Statements

The COPY, ++INCLUDE, and REPLACE statements allow you to modify the program's source text at compile time. Conditional compiling statements such as $IF and $SET statements can be used to specify whether or not certain lines of code are compiled or skipped. See **Section 2.5, "Conditional Compilation"** for details on these statements.

2.4.1 COPY Statement

The COPY statement copies text or a resource (static data such as a bitmap) into the source program from the specified file immediately prior to compilation. The text or resource is *inserted* for compilation only and does not permanently replace the COPY statement in the program source. Resources and COPY files that are inserted in this way into the object code are loaded from the object file at runtime. If you change the resource (such as a bitmap) or the COPY file, you must recompile for the change to be reflected in the object code.

The REPLACING phrase allows word and substring substitutions to be made in the inserted text prior to compilation.

Note: This manual entry includes code examples and highlights for first-time users following the General Rules section.

General Format

Format 1

```
COPY INDEXED library-name [ {IN} path-name ] [ SUPPRESS ]
                           {OF}

[ REPLACING { {  old-text BY new-text             } } ... ] .
            { { {LEADING } literal-1 BY {literal-2} } }
            { { {TRAILING}              {SPACE    } } }
            { {                         {SPACES   } } }
```

Format 2

```
COPY RESOURCE resource-name [  {IN} path-name ] .
                               {OF}
```

Syntax Rules

1. The COPY statement must be terminated by a period. The period is part of the COPY statement and does not otherwise affect the program.

2. *Library-name* must be a nonnumeric literal or user-defined word. *Path-name* must be a nonnumeric literal or a user-defined word. Note that a nonnumeric literal may reference an environment variable by placing a "$" in the name, as described in General Rule 2. To preserve the case of *library-name* and *path-name*, you must place them within quotation marks, otherwise they will be treated as uppercase by case sensitive operating systems. For more information, see the *ACUCOBOL-GT User's Guide*, **section 2.6, "COPY Libraries."**

3. The COPY statement may be used anywhere a separator may occur. It may be placed in Area A or Area B.

4. Old-text and new-text may be any of the following:

   a. A series of text words placed between "==" delimiters. For example "==WORD-1 WORD-2==" specifies a two-word sequence. In *old-text*, at least one word must be specified. In *new-text*, zero words may be used.

   b. A numeric or nonnumeric literal.

   c. A data name, including qualifiers, subscripts, and reference modification.

   d. Any single text word.

5. For purposes of the COPY statement, a "text word" is a contiguous sequence of characters in Area A or Area B that form one of the following:

   a. A separator, except for: space, a pseudo-text delimiter ("=="), and the opening and closing delimiters for nonnumeric literals.

   b. A numeric or nonnumeric literal.

   c. Any of a sequence of characters except comment lines and the word "COPY", bounded by separators, which is neither a separator nor a literal.

6. *Literal-1* and *literal-2* are nonnumeric literals.

7. The phrases SPACE and SPACES are equivalent. When one of these is used instead of *literal-2*, *literal-1* is deleted and no spaces are actually substituted.

8. The format of the COPY file must conform to one of the allowed ACUCOBOL-GT source formats (either terminal or ANSI). This format need not be the same as that used in the rest of the program. Book 1, section 2.5, contains details about which source format is used for COPY files.

9. *Resource-name* must be an alphanumeric literal or a user-defined word. A resource name with a hyphen is equivalent to the same name with an underscore in place of the hyphen. For example, "MY-FILE" is treated as being identical to "MY_FILE". To preserve the case of *resource-name*, you must place it within quotation marks; otherwise, it will be treated as uppercase by case-sensitive operating systems.

10. COPY statements may be nested in other COPY libraries. Any one of the COPY statements in this structure can include the REPLACING phrase.

    Depending on the scope of each statement, the REPLACING phrases might affect subsidiary COPY statements. For example, if "program-a.cbl" contains a copy/replace as follows:

    ```
    COPY "program-b.cpy"
       REPLACING ==genericitems== BY ==myitems==.
    ```

    and "program-b.cpy" contains a nested copy/replace statement:

    ```
    COPY "program-c.cpy"
       REPLACING ==variabledata== BY == specificdata==.
    ```

    The replace performed in "program-b.cpy" by the copy/replace statement in "program-a.cbl" will affect "program-c.cpy." If you do not want the copy/replace statement in "program-a.cbl" to cascade to "program-c.cpy", you must add the following statement to "program-b.cpy", so that the copy/replace performed in "program-b.cpy" will not be performed in "program-c.cpy."

    ```
    COPY "program-c.cpy"
       REPLACING ==genericitems== BY ==genericitems==.
    ```

General Rules

1. *Library-name* and *path-name* identify a source file to be included at the location of the COPY statement. The text of the source file logically replaces the COPY statement, including the terminating period. The

rules for interpreting these names are described in Book 1, **Section 2.6**. The "-Ce" compile option can be used to specify an alternate default filename extension. See Book 1, **Section 2.2.5**.

2. You may use operating system environment variables in the OF phrase of a COPY statement. To reference an environment variable, place a "$" in front of it. For example, if you assign "MYLIB" to "C:\MYFILES\MYLIB", then the statement:

   ```
   COPY "FILE1" OF "$MYLIB"
   ```

   would use the file C:\MYFILES\MYLIB\FILE1".

   You may use multiple environment variables by preceding each one with a $ symbol. Symbol names may contain alphanumeric characters, hyphens, underscores, and dollar signs. If the symbol name is not found in the environment, then it is left unchanged (including the initial $ symbol). Symbols are not processed recursively--if the value of a symbol contains a $, the dollars sign is used literally in the final file name.

3. When INDEXED appears after the word COPY, it is ignored by the compiler. It may be included to provide compatibility with some older COBOL dialects.

4. If the word SUPPRESS appears after *library-name* and *path-name*, then the program listing file will not include the contents of the COPY file or any other COPY files that may be nested within. This word provides compatibility with one feature of IBM DOS/VS COBOL. It is not a reserved word in ACUCOBOL-GT and may be used in other contexts as a user-defined name.

5. The text of the COPY file is copied unchanged into the source program unless the REPLACING option is used. If the REPLACING option is used, then elements of the COPY file that match *old-text* or *literal-1* are replaced by *new-text* or *literal-2*. The comparison operation that determines text replacement is done as follows:

   a. The leftmost library text word that is not a separator comma or semicolon is the first text word used for comparison. Starting with this word, and the first *old-text* specified, the entire *old-text* sequence is compared with an equivalent number of contiguous library text words.

b. *Old-text* matches the library text only if the ordered sequence of text words of *old-text* is identical to the ordered sequence of library text words. For purposes of matching, a separator semicolon, comma, or space is considered a space, and a sequence of one or more spaces is considered a single space. Also, lower-case characters are considered the same as upper-case characters in all text words except for nonnumeric literals.

c. If no match occurs, the comparison is repeated for each *old-text* specified until a match is found or each *old-text* has been tried.

d. After all *old-text* comparisons have been tried and no match has occurred, the leftmost library text word is copied into the source program. The next text word is then considered as the leftmost word and the cycle is repeated.

e. Whenever a match occurs between the library text and *old-text*, the corresponding *new-text* is placed in the source program. The library text word that follows the rightmost word that participated in the match then becomes the new leftmost word for subsequent cycles.

f. When you are using the LEADING/TRAILING option, the replacement process differs slightly. When a match occurs between library text and *literal-1*, the only characters replaced by *literal-2* are the specific LEADING or TRAILING characters indicated in the COPY statement. These characters can be a substring or a whole word. If a SPACE or SPACES phrase is used, the LEADING or TRAILING characters are deleted. For example, if you have the following COPY library named "MY-COPY.CPY":

```
01 dummy-rec.
    03 dummy-number-null    PIC X(10)
```

and you used this COPY statement:

```
COPY "MY-COPY.CPY" REPLACING
    LEADING "dummy" by "employee"
    TRAILING "null" by SPACES.
```

Then the replacement will result in:

```
01 employee-rec.
```

```
03 employee-number    PIC X(10)
```

Note that when using ALPHANUMERIC strings such as "02" in the LEADING BY phrase, it is best to use the == delimiters rather than surrounding with quotes.

g. The comparison cycle continues until the rightmost text word in the library has either participated in a match or has been the leftmost word of a comparison cycle.

6. Comment lines and blank lines occurring in the library or in *old-text* are ignored for purposes of matching. Comment lines and blank lines occurring in library text that is matched by a REPLACING operand are not copied into the source program.

7. Debugging lines may appear within the library text and in *old-text*. Text words appearing in a debugging line participate in the matching rules as if the line were a normal text line.

8. When *new-text* is copied into the source program, the first word of *new-text* is copied into the same Area as the leftmost word of the replaced text. Subsequent words of *new-text* are copied into Area B.

9. It is possible to use the REPLACING phrase to replace substrings. This allows you to construct COPY libraries in which several strings have a uniform substring that you plan to modify.

   For example, the substring “individual” might occur in the COPY library in “individual”-name, “individual”-address, “individual”-state, “individual”-city, “individual”-zip, and “individual”-title. The REPLACING phrase could be used to replace “individual” with specific substrings such as employee, owner, student, teacher, professor, or advisor.

   To make use of this, delimit the substring that will be replaced in the COPY library with quotes. Then use the standard COPY syntax to replace the quoted substring by another substring. The resulting sequence of characters is re-evaluated by the compiler to make a new string.

   For example, suppose you have a COPY library (called “MYLIB”) that contains the following:

```
77  MY-'DUMMY'-DATA-ITEM  PIC X(10).
```

and you used this COPY statement:

```
COPY "MYLIB" REPLACING ==':DUMMY:'== BY ==REAL==.
```

Then the text of “MYLIB” is effectively treated as:

```
77  MY-REAL-DATA-ITEM  PIC X(10).
```

You should use hyphens rather than underscores in this instance.

In addition to the use of single and double quotes to delimit the substring, the following delimiters are also allowed:

```
==(XYZ)==
==|XYZ|==  (in HP COBOL compatibility mode)
==*XYZ*==
==:XYZ:==
==XYZ*==
==XYZ&==
==XYZ#==
```

10. *Resource-name* and *path-name* identify a resource file to be included in the resulting object file. The rules for interpreting these names are described in Book 1, **Section 2.6**. Note that the compiler’s “COPY path” applies to resources (Format 2) as well as to source files (Format 1).

11. The effect of a COPY RESOURCE statement is to add *resource-name* to a list of resources that the compiler embeds into the resulting COBOL object file. The resources are added to the end of the COBOL object in the same order as the corresponding COPY statements. Because the resources are added to the end of the object, the location of the corresponding COPY RESOURCE statement in the COBOL program is irrelevant. Conventionally, COPY RESOURCE statements are placed either in Working-Storage or at the end of the program, but any location is acceptable.

12. If *resource-name* resolves to a COBOL object or library file, the compiler includes this object or library in the resulting object in a manner similar to “cblutil -lib”. These are not considered resources, but are embedded COBOL objects. Note that we recommend using “cblutil -lib” to create libraries containing multiple COBOL objects instead of using COPY RESOURCE. There are two advantages to using “cblutil”. The first is that you do not need to worry about the

order in which COBOL objects are compiled (if you use COPY RESOURCE, you must ensure that the copied object is compiled first), and “cblutil” also checks for duplicated program names; COPY RESOURCE does not.

Code Examples

Assume the existence of disk directory CODELIB. In directory CODELIB is file ENROLLREC. The contents of ENROLLREC are:

```
01  ENROLLMENT-RECORD.
    05 STUDENT-NAME     PIC X(30).
    05 STUDENT-ADDR     PIC X(50).
    05 STUDENT-GPA      PIC 99V9.
    05 SID              PIC 9(7).
```

Code example 1:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  COPY-EXAMPLE-1.
...
DATA DIVISION.
FILE SECTION.
FD  SCIENCE-DEPT-ENROLLMENT-FILE.
COPY ENROLLREC IN "LIBRARY/CODELIB".
...
```

Code compiled after COPY substitutions:

```
...
DATA DIVISION.
FILE SECTION.
FD  SCIENCE-DEPT-ENROLLMENT-FILE.
01  ENROLLMENT-RECORD.
    05 STUDENT-NAME     PIC X(30).
    05 STUDENT-ADDR     PIC X(30).
    05 STUDENT-GPA      PIC 99V9.
    05 SID              PIC 9(7).
...
```

Code example 2:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  COPY-EXAMPLE-2.
...
DATA DIVISION.
```

```
FILE SECTION.
FD  SCIENCE-DEPT-ENROLLMENT-FILE.
COPY ENROLLREC
   IN "LIBRARY/CODELIB"
      REPLACING ==SID==  BY ==STUDENT-ID==,
                ==9(7)== BY ==9(9)==.
...
```

Compiled code after COPY/REPLACING substitutions:

```
...
DATA DIVISION.
FILE SECTION.
FD  SCIENCE-DEPT-ENROLLMENT-FILE.
01  ENROLLMENT-RECORD.
    05  STUDENT-NAME     PIC X(30).
    05  STUDENT-ADDR     PIC X(30).
    05  STUDENT-GPA      PIC 99V9.
    05  STUDENT-ID       PIC 9(9).
...
```

Highlights for first-time users

1. COPY will always import the *entire* contents of the named COPY file.
2. The REPLACING text does *not* appear in the listing produced by the ACUCOBOL-GT compiler ("-Lo *filename*" compiler argument). This is a common source of confusion for users who check the compilation listing file for verification that the replacing action occurred. You can, however, use the "-Lp" compiler option to create an output file that includes the REPLACING text. See "-Lp" in **Section 2.2.3** of Book 1.

2.4.2 ++INCLUDE Statement

To provide more compatibility with other COBOLs, ACUCOBOL-GT also supports the ++INCLUDE statement. This statement is very similiar in function, format, and syntax to the COPY statement described in **Section 2.4.1** The differences from the COPY statement are:

- A terminating period is not required in ++INCLUDE.
- ++INCLUDE can refer to only a single file, not to a library.

- ++INCLUDE does not allow SUPPRESS.
- ++INCLUDE does not allow NOLIST.
- ++INCLUDE does not allow REPLACING.

Filenames referenced by ++INCLUDE are searched for in the same way as COPY files.

2.4.3 REPLACE Statement

The REPLACE statement provides the ability to modify source text selectively. Text replacement is accomplished by the compiler immediately prior to source compilation.

REPLACE is frequently used to help facilitate single source code maintenance across multiple COBOL versions or multiple hardware or operating system environments. REPLACE may be used wherever there is a need to make temporary text substitutions for compilation purposes.

Note: This manual entry includes code examples and highlights for first-time users following the General Rules section.

General Format

Format 1

```
REPLACE  {  { old-text BY new-text               }  } ... ] .
         {  { {LEADING } literal-1 BY {literal-2} }  }
         {  { {TRAILING}              {SPACE    } }  }
         {  {                         {SPACES   } }  }
```

Format 2

```
REPLACE OFF.
```

Syntax Rules

1. The REPLACE statement must be terminated by a period. The period is part of the REPLACE statement and does not otherwise affect the program.
2. The REPLACE statement may be used anywhere a separator may occur. It may be placed in Area A or Area B.
3. *Old-text* and *new-text* may be any of the following:
   a. A series of text words placed between "==" delimiters. For example "==WORD-1 WORD-2==" specifies a two-word sequence. In old-text, at least one word must be specified. In new-text, zero words may be used.
   b. A numeric or nonnumeric literal.
   c. A data name, including qualifiers, subscripts, and reference modification.
   d. Any single text word.
4. For purposes of the REPLACE statement, a "text word" is a contiguous sequence of characters in Area A or Area B that form one of the following:
   a. A separator, except for: space, a pseudo-text delimiter ("=="), and the opening and closing delimiters for nonnumeric literals.
   b. A numeric or nonnumeric literal.
   c. Any of a sequence of characters except comment lines and the word "COPY," bounded by separators, which is neither a separator nor a literal.
5. *Literal-1* and *literal-2* are nonnumeric literals.
6. The phrases SPACE and SPACES are equivalent. When one of these is used instead of *literal-2*, *literal-1* is deleted and no spaces are actually substituted.

General Rules

1. The REPLACE statement specifies conversion of source statements containing *old-text* into *new-text*. The scope of a REPLACE statement continues from the first text word following the REPLACE statement to the beginning of the next REPLACE statement, or the end of the program. A Format 2 REPLACE statement terminates the scope of any preceding REPLACE statement.

2. REPLACE statements are processed after COPY statements. The text produced by the action of a REPLACE statement must not contain a REPLACE statement.

3. Within the scope of a REPLACE statement, any source text that matches *old-text* is logically replaced by *new-text*. The comparison operation that determines text replacement is done as follows:

   a. The leftmost source text word is the first text word used for comparison. Starting with this word, and the first old-text specified, the entire old-text sequence is compared with an equivalent number of contiguous source text words.

   b. Old-text matches the source text only if the ordered sequence of text words of old-text is identical to the ordered sequence of source text words. For purposes of matching, a separator semicolon, comma, or space is considered a space, and a sequence of one or more spaces is considered a single space. Also, lower-case characters are considered the same as upper-case characters in all text words except for nonnumeric literals.

   c. If no match occurs, the comparison is repeated for each old-text specified until a match is found or each old-text has been tried.

   d. After all old-text comparisons have been tried and no match has occurred, the next source text word is then considered as the leftmost word and the cycle is repeated.

   e. Whenever a match occurs between the source text and old-text, the corresponding new-text replaces old-text in the source program. The source text word that follows the rightmost word that participated in the match then becomes the new leftmost word for subsequent cycles.

f. When you are using the LEADING/TRAILING option, a match between library text and *literal-1* will replace only the specific LEADING or TRAILING characters indicated in the REPLACE statement with the text in *literal-2*. These characters can be a substring or a whole word. If a SPACE or SPACES phrase is used, the LEADING or TRAILING characters are deleted.

g. The comparison cycle continues until the rightmost text word in the REPLACE scope has either participated in a match or has been the leftmost word of a comparison cycle.

4. Comment lines and blank lines occurring in *old-text* are ignored for purposes of matching.

5. Debugging lines may appear in *old-text*. Text words appearing in a debugging line participate in the matching rules as if the line were a normal text line.

6. When *new-text* is copied into the source program, the first word of *new-text* is copied into the same Area as the leftmost word of the replaced text. Subsequent words of *new-text* are copied into Area B.

7. It is possible to use the REPLACE statement to replace substrings. In addition to the use of single and double quotes to delimit the substring, the following delimiters are also allowed:

```
==(XYZ)==
==*XYZ*==
==:XYZ:==
```

Code Examples

```
REPLACE
   ==STANDARD-ALPHA==    BY   ==ALPHA-UPPER-CASE==
   ==TABLE-SIZE==        BY   ==MAX-TABLE-SIZE==
   ==PAGE-BUFFER-SIZE==  BY   ==SHORT-PAGE-SIZE==
   ==WITH-DEBUG-MODE==   BY   ====.
*delete matched text
...
REPLACE OFF.
*turns off REPLACE
```

Highlights for first-time users

1. Multiple REPLACE statements are permitted. The REPLACE statement can appear anywhere in the program source.
2. The substitution actions of the REPLACE statement continue to affect the program source until the REPLACE statement is either superseded by a new REPLACE statement or turned off by the REPLACE OFF statement.
3. REPLACE statements are processed after COPY statements.
4. REPLACE statements can not contain COPY statements. COPY statements may contain REPLACE statements.
5. The replaced text does *not* appear in the listing produced by the ACUCOBOL-GT compiler ("-Lo *filename*" compiler argument). This is a common source of confusion for users who check the compilation listing file for verification that the replacing action occurred. You can, however, use the "-Lp" compiler option to create an output file that includes the replaced text. See "-Lp" in section 2.1.3 of Book 1.

2.5 Conditional Compilation

Conditional compilation provides a mechanism for selectively compiling part or all of the COBOL source. Conditional compilation is controlled by $IF, $ELSE, and $END constructs, which behave in a similar way to the COBOL IF construct. Conditional compilation also supplies the $DISPLAY Statement, which can be used to display a message during compilation or include a version number in the object file. The $SET statement can be used to define compiler directives for use in $IF statements.

Note: The compiler has special conditional compilation options that turn on compiler directives and set constants to values. There is also a compiler option (-Cg) that turns of conditional compiling features. See the *ACUCOBOL-GT Users Guide*, **section 2.2.15** for details.

Syntax Rules

1. Conditional compilation statements are indicated by a dollar sign ($) in the indicator area of the source line followed by one of the key words IF, DISPLAY, ELSE, END, and SET.
2. Conditional compilation should not be used to split a COBOL character string; that is, continuation lines should not be split by conditional compilation controls.

2.5.1 $DISPLAY Statement

The $DISPLAY statement displays a message on the standard output device during compilation, or includes a version number in the object. There are two formats.

Format 1

```
$DISPLAY text-data
```

Format 2

```
$DISPLAY VCS = version-number
```

Syntax

1. The entire $DISPLAY statement must appear on a single line.

General rules

1. If a $DISPLAY statement is encountered on a source line that is ignored by conditional compilation, there is neither a compile-time nor a runtime effect.

   Format 1

2. Text-data is displayed on the standard output device during compilation. There is no runtime effect.

   Format 2

3. Version-number is the content of the entire source line following the "=", excluding leading and trailing spaces.

4. The character string formed by concatenating "@(#)", version-number, and a null character (binary zero) is included in the object file. If version-number begins with the characters "@(#)", the compiler does not concatenate these characters when forming the character string. In other words, only a single "@(#)" will be included in the object file, whether version-number includes that string or not.

Note: Version-number can be any text string, but it is intended to contain a version number for which a pattern matching tool, such as the UNIX sccs "what" command, can search the object file.

2.5.2 $END Statement

The $END Statement is used in conjunction with the **$IF Statement** to control conditional compilation. There is a single format:

$END

Syntax

1. The whole statement must appear on a single line.

General rules

1. The innermost $IF statement is terminated. The now active $IF condition is considered. If the active condition is "true", the source lines following the $END are processed. If the condition is "false", COBOL source lines are ignored until the next conditional compilation line is encountered.

2.5.3 $ELSE Statement

The $ELSE Statement is used in conjunction with the **$IF Statement** to control conditional compilation. There is a single format:

$ELSE

Syntax

1. The whole statement must appear on a single line.

General rules

1. The most recent $IF condition is reversed. If the now active $IF condition is "true", the source lines following the $ELSE are processed. If the $IF condition is "false", COBOL source lines are ignored until the next conditional compilation line is encountered.

2.5.4 $IF Statement

The $IF Statement provides the ability to conditionally include or exclude text based on the state of certain variables. There are three formats:

Format 1

```
$IF constant-name-1 [NOT] {< > =} literal-1
```

Format 2

```
$IF constant-name-2 [NOT] DEFINED
```

Format 3

```
$IF directive-setting SET
```

Syntax

1. Constant-name-1 is defined by a level 78 item or a CONSTANT compiler flag.
2. Directive-setting is specified in the same format as it is given in the $SET statement and may be preceded by NO. However, the format used in the $IF statement differs from the format used in the $SET statement as follows:
    - No spaces are allowed between the NO and the directive name

- Case must be preserved in the directive

Directive-setting may also be specified at compile time by using the "-/" (forward slash) compiler option. See the User's Guide, **Section 2.2.15** for details on this and other conditional compiler options.

3. The entire statement must appear on a single line.
4. $IF can be nested within another $IF.

General rules

1. constant-name-2 is DEFINED if it is the name of a level 78 item or a CONSTANT compiler flag. Otherwise it is NOT DEFINED.
2. Directive setting SET evaluates "true" if the given string matches the actual directive setting.
3. The comparison between directive-setting and the actual directive is case-sensitive.
4. If the condition evaluates "true", the source lines following the $IF statement are processed. If the condition evaluates "false", COBOL source lines are ignored until the next conditional compilation line is encountered.

2.5.5 $SET Statement

The $SET statement can be used to define compiler directives for use in **$IF Statement**. The directives set with this method have no value, they are only set. The $SET statement can also be used to set values for COBOL variables in the same way as a level 78 data item.

There are two formats for the $SET statement.

Format 1

```
$SET [NO]compiler-directive
```

Format 2

```
$SET CONSTANT identifier value
```

Syntax

1. The whole statement must appear on a single line.

General rules

Format 1

1. The sole effect of the Format 1 $SET statement is to set a directive name in the compilation unit. Alternatively, you can use the “-/” (forward slash) compiler option to set a directive name. Refer to the ACUCOBOL-GT User’s Guide, **Section 2.2.15**, Conditional Compiling Options for details. If a later Format 3 $IF statement is encountered, this statement will evaluate "true" if the compiler-directive set by $SET matches the directive-setting in the $IF statement. Preceding the compiler directive with NO turns off the setting.

Format 2

2. "identifier" must be a valid COBOL identifier.
3. "value" is any valid value for a COBOL identifier.
4. If "value" is surrounded by single (') or double (") quotes, it is a string literal. "identifier" can be used anywhere a string literal can be used.
5. If "value" is surrounded by parentheses (()), it is a numeric literal.
6. "identifier" can be used anywhere a numeric literal can be used.
7. If "value" is neither quoted nor surrounded by parentheses, it will be considered a numeric literal if all of the characters in the value are digits. Otherwise it is considered a string literal.
8. A format 2 $SET statement is equivalent to the following line:

   ```
   77 identifier VALUE value.
   ```

   with the exception that the level "77" line doesn't allow numeric values to be surrounded with parentheses.

2.6 Program Organization

A COBOL program is divided into four parts, called divisions. The divisions are the Identification Division, the Environment Division, the Data Division, and the Procedure Division. Divisions can contain sections, which in turn can contain paragraphs. Paragraphs are composed of sentences, clauses, statements and entries.

The general format of a program is:

```
[ identification division ]

[ environment division ]

[ data division ]

procedure division

[ END PROGRAM string_literal. ]
```

Each of these divisions is described in detail in the following chapters. The END PROGRAM statement is optional. If used, it must appear in Area A.

2.6.1 Program Elements

A program can consist of many types of elements. These elements are described in the following sections.

2.6.1.1 Division header

A division header names and marks the beginning of a division. The formats for particular divisions are described in the appropriate syntax charts. Division headers must appear in Area A.

The following division headers are optional and may be included or excluded at your option. The compiler determines which division it is processing by other COBOL syntax.

- Identification Division

- Environment Division
- Data Division

2.6.1.2 Section header

A section header marks the beginning of a section in the Environment, Data, and Procedure Divisions. In the Environment and Data Divisions, the formats of the allowed section headers are described in the appropriate syntax charts in Chapter 4 and Chapter 5, respectively. In the Procedure Division, a section header is a user-defined word followed by the word "SECTION" and an optional segment number. A period always follows a section header. Section headers must appear in Area A.

2.6.1.3 Paragraph header

A paragraph header names a paragraph in the Identification, Environment, and Procedure Divisions. In the Identification and Environment Division, a paragraph header is a reserved word followed by a period. These are detailed in the appropriate syntax charts.

In the Procedure Division, a paragraph header is a user-defined word followed by a period. A paragraph header can be placed in Area A or Area B. When a paragraph header is placed in Area B, the compiler produces the following warning message, unless the "-w" compile switch is specified:

```
Warning: Paragraph Name found in Area B
```

An initial paragraph header, immediately following the Procedure Division header, is not required. If no initial paragraph header is present, the compiler creates a dummy header named "ACU-MAIN".

The first entry or sentence of a paragraph begins on either the same line as the paragraph header or in Area B of succeeding lines. Subsequent entries or sentences must be in Area B.

2.6.1.4 Clauses and entries

An entry is an item of descriptive nature composed of separate clauses. Each clause specifies some attribute of its entry. Clauses are separated by spaces (or commas or semicolons). An entry is terminated by a period. The format of clauses and entries is described in the appropriate syntax diagrams.

2.6.1.5 Statements

A statement is a COBOL key word (called a verb) followed by its operands. A statement directs either the compiler or the object program to take some action. There are four types of COBOL statements:

1. *Compiler-directing* statements specify an action to be taken by the compiler. Only COPY, REPLACE, and USE statements fit this classification.

2. *Imperative* statements specify an unconditional action to be taken by the object program at run time. Whenever an imperative statement is allowed, it may consist of a sequence of consecutive imperative statements.

3. *Conditional* statements specify an action to be taken by the object program that is dependent on the truth value of some condition.

4. *Delimited-scope* statements specify their explicit scope terminator. This scope terminator always has "END-" as the first four letters of its word. A delimited-scope statement contains elements of a conditional nature. Because of the scope-delimiter, however, these statements may be used anywhere an imperative statement may be.

2.6.1.6 Sentences

A sentence is a sequence of one or more statements terminated by a period. An *imperative* sentence is one that contains only imperative statements. A *conditional* sentence consists of a conditional statement optionally preceding a sequence of imperative statements.

3 Identification Division

Key Topics

3.1 Identification Division

The Identification Division marks the beginning of a COBOL program. It serves to name and comment the program.

General Format

```
[ IDENTIFICATION DIVISION. ]

[ ID DIVISION.             ]

[ PROGRAM-ID. program-name [ IS {INITIAL } PROGRAM ] . ]
                                {RESIDENT}

[ AUTHOR.    [ comment-entry ] ... ]

[ INSTALLATION.   [ comment-entry ] ... ]

[ DATE-WRITTEN.   [ comment-entry ] ... ]

[ DATE-COMPILED.  [ comment-entry ] ... ]

[ SECURITY.  [ comment-entry ] ... ]

[ REMARKS.  [ comment-entry ] ... ]
```

Syntax Rules

1. A comment-entry can consist of any set of characters over any number of lines. It ends with the next line that starts in Area A.
2. “ID DIVISION” is interchangeable with “IDENTIFICATION DIVISION”.
3. The AUTHOR, INSTALLATION, DATE_WRITTEN, DATE_COMPILED, SECURITY, and REMARKS paragraphs may be placed in any order.

General Rule

The AUTHOR, INSTALLATION, DATE-WRITTEN, DATE-COMPILED, and SECURITY paragraphs are used solely for commentary.

3.2 PROGRAM-ID Paragraph

General Format

```
[ PROGRAM-ID.  program-name [ IS {INITIAL } PROGRAM ] . ]
                                 {RESIDENT}
```

Syntax Rule

Program-name is a user-defined word or a reserved word. It must be unique among separately compiled programs. If a reserved word is used, it is treated as if it were not reserved. The maximum number of characters is 30.

General Rules

1. The *program-name* identifies the name of the program. It is used by the ACUCOBOL-GT runtime system and debugger to identify a program.

2. The INITIAL PROGRAM clause specifies an initial program. Whenever an initial program is called, it is placed in its initial state. Data contained in an initial program is set to its starting value every time the program is called. Note that the "-Zi" compiler option causes the program to be compiled as if it had the IS INITIAL PROGRAM phrase specified. See Book 1, **Section 2.2.16, "Miscellaneous Options"** for details on the -Zi compiler option.

3. Files contained in the program are not in the open mode:

   a. the first time the program is called,

   b. the first time the program is called after it has been the target of a CANCEL statement,

   c. every time the program is called if it is an INITIAL program.

4. On all other entries, the files contained in the program are in the same state and position as when the program last exited.

5. INITIAL programs are removed from memory when they exit. Non-initial programs remain in memory until they are the targets of a CANCEL statement.

6. The RESIDENT clause specifies that the program is to remain resident in memory after its first execution. A program with the RESIDENT clause cannot be affected by a CANCEL statement. Note that the RESIDENT clause shields selected programs from the effects of a **CANCEL Statement.**

7. The IBM DOS/VS COBOL "-Cv" compatibility mode allows the name to be enclosed in quotation marks. See the IBM DOS/VS COBOL chapter in *Transitioning to ACUCOBOL-GT* for more information.

8. If you omit the PROGRAM-ID paragraph, the program's name is derived from the source file name as follows:

   a. All directory information is removed from the file name and only the base file name is used.

   b. If the file name includes a period (".") or space, that portion of the file name (including the period or space) is truncated.

   c. The name is translated to upper case for letters in the ASCII alphabet range.

   d. Characters not in the ASCII alphabet are left unchanged.

9. If the PROGRAM-ID paragraph is omitted, the program does not have an INITIAL or RESIDENT state.

4 Environment Division

Key Topics

4.1 Environment Division

The Environment Division describes the program's physical environment, primarily through the descriptions of the files it uses.

General Format

```
[ [ ENVIRONMENT DIVISION. ]

[ [ CONFIGURATION SECTION. ]

[ SOURCE-COMPUTER.  source-computer-entry ]

[ OBJECT-COMPUTER.  object-computer-entry ]

[ SPECIAL-NAMES.  [ special-names-entry ] ] ]

[ [ INPUT-OUTPUT SECTION. ]

  [ FILE-CONTROL. ]  { file-control-entry } ...

[ I-O-CONTROL.  [ i-o-control-entry ] ] ] ]
```

Syntax Rule

The division header is optional for the Environment Division.

General Rule

The Environment Division entries are described in the following sections.

4.2 Configuration Section

The Configuration Section contains information about the machine environment for the program. The section header for the Configuration Section is optional.

4.2.1 Source-Computer Paragraph

The SOURCE-COMPUTER paragraph identifies the computer on which the source program is compiled.

General Format

```
SOURCE-COMPUTER.  computer-name

   [ WITH DEBUGGING MODE ] .
```

Syntax Rule

Computer-name is a user-defined word, or multiple words separated by spaces, that names the source computer.

General Rules

1. *Computer-name* is for documentation purposes only.
2. If the WITH DEBUGGING MODE clause is used, then conditional debugging lines in the source program will be treated as standard source lines, not comment lines. This can also be accomplished by compiling with the "-Sd" compiler option.

4.2.2 Object-Computer Paragraph

The OBJECT-COMPUTER paragraph names the computer on which the program is to be run.

General Format

```
OBJECT-COMPUTER.  computer-name

   [ MEMORY SIZE integer {WORDS     } ]
                         {CHARACTERS}
                         {MODULES   }

   [ PROGRAM COLLATING SEQUENCE IS alphabet-name ]

   [ SEGMENT-LIMIT IS seg-val ] .
```

Syntax Rules

1. *Computer-name* is a user-defined word, or multiple words separated by spaces, that names the object computer.
2. *Alphabet-name* is a user-defined word that describes the collating sequence of the object machine.
3. *Seg-val* is an integer literal between 1 and 49 inclusive.

General Rules

1. *Computer-name* is for documentation purposes only.
2. The MEMORY SIZE clause is for documentation only.
3. The COLLATING SEQUENCE clause specifies the collating sequence for any alphanumeric comparisons done in the program. *Alphabet-name* must be an alphabet described in SPECIAL-NAMES. The sequence of the characters in the alphabet determines the sequence for character comparisons. It also specifies the default collating sequence for SORT and MERGE verbs. The character that is first in the program collating sequence is treated as the LOW-VALUES character for the program. The character that is last in the program collating sequence is treated as the HIGH-VALUES character for the program. (The one exception to this is that in Special-Names, LOW-VALUES and HIGH-VALUES always refer to the first and last characters in the native collating sequence.)
4. The SEGMENT-LIMIT clause defines which Procedure Division sections are to be placed in overlays. If the SEGMENT-LIMIT clause is not specified, then any section with a segment number of 50 or greater is placed in overlays. When the SEGMENT-LIMIT clause is used, then any section with a segment number of *seg-val* or greater is placed in overlays.

4.2.3 Special-Names Paragraph

The SPECIAL-NAMES paragraph describes several miscellaneous aspects of the operating environment. The phrases may be listed in any order, with two exceptions. The switch declarations must come first, and alphabets must be defined before they are referenced in SYMBOLIC CHARACTERS phrases.

General Format

```
SPECIAL-NAMES.

   [ {switch-name} [ IS mnemonic-name ]
     {system-name}

      [ {ON } STATUS IS cond-name ] ... ] ...
        {OFF}

   [ {alphabet-entry } ... ]

   [ SYMBOLIC CHARACTERS

      { {name} ... {IS } {number} ... } ...
                   {ARE}

      [ IN alphabet-name ] ]

   [ CLASS class-name IS

      { lit-1 [  {THROUGH} lit-2 ] } ... ] ...
                 {THRU   }

   [ CURRENCY SIGN IS char ]

   [ DECIMAL-POINT IS COMMA ]

   [ NUMERIC SIGN IS TRAILING SEPARATE ]

   [ CONSOLE IS CRT ]

   [ CURSOR IS cursor-name ]

   [ CRT STATUS IS status-name ]
```

```
[ SCREEN CONTROL IS control-name ]

[ EVENT STATUS IS event-status ].
```

Alphabet Entry

Format 1

```
ALPHABET alphabet-name IS (STANDARD-1}
                          {STANDARD-2}
                          {NATIVE    }
                          {EBCDIC    }
```

Format 2

```
ALPHABET alphabet-name IS
   {literal-1 [ THROUGH literal-2    ] } ...
              [ THRU literal-2       ]
              [ {ALSO literal-3}  ...]
```

Syntax Rules

1. *Switch-name* must be one of the system names: SWITCH-1, SWITCH-2, ... SWITCH-26 *or* the word “SWITCH” followed by a switch number (a numeric literal 1 through 26 or an alphanumeric literal “A” through “Z”). It represents one of the 26 program switches.

2. *Mnemonic-name* is a user-defined word that may be used in a SET statement to change the state of the associated program switch or to refer to a device in an ACCEPT or DISPLAY statement.

3. Each *system-name* must be associated with a *mnemonic-name*. Also, no *system-name* may be given an ON or an OFF STATUS. *System-name* must be one of the following: CONSOLE, SYSIN, SYSIPT, SYSOUT, SYSLIST, SYSLST, SYSOUT-FLUSH, or SYSERR.

4. *Cond-name* is a user-defined word that can be used to test the status of a program switch.

5. For each *switch-name*, at least one *mnemonic-name* or one *cond-name* must be specified. No more than one ON STATUS and one OFF STATUS phrase may be specified for a particular *switch-name*.

6. *Name* is a user-defined word that names a symbolic character.

7. *Number* is an integer literal that must be in the range of ordinal positions in the alphabet being referenced.

8. There must be a one-to-one correspondence between occurrences of *name* and *number.* The relationship between each *name* and *number* is by position in the SYMBOLIC CHARACTERS clause. The first *name* is paired with the first *number*, the second with the second, and so on.

9. *Class-name* is a user-defined word that defines a class name.

10. *Lit-1* and *lit-2* are numeric or alphanumeric literals.

11. *Char* is a one-character nonnumeric literal that specifies a currency symbol.

12. *Cursor-name* must be the name of a data item appearing in the Data Division that is 4 or 6 characters in length. *Cursor-name* must describe an elementary unsigned numeric integer or a group item containing two such elementary data items.

13. *Status-name* must name a group item in the Data Division that is three characters in length *or* must name an elementary numeric data item.

14. *Control-name* must name a group item with the following structure:

```
01 SCREEN-CONTROL.
   03 ACCEPT-CONTROL   PIC 9.
   03 CONTROL-VALUE    PIC 999.
   03 CONTROL-HANDLE   USAGE HANDLE.
   03 CONTROL-ID       PIC X(2) COMP-X.
```

    You must use the preceding structure, but you may use your own names for the variables.

15. *Event-status* must refer to a group item with the following structure:

```
01 EVENT-STATUS.
   03 EVENT-TYPE           PIC X(4) COMP-X.
   03 EVENT-WINDOW-HANDLE  USAGE HANDLE OF WINDOW.
   03 EVENT-CONTROL-HANDLE USAGE HANDLE.
   03 EVENT-CONTROL-ID     PIC X(2) COMP-X.
   03 EVENT-DATA-1         USAGE SIGNED-SHORT.
   03 EVENT-DATA-2         USAGE SIGNED-LONG.
   03 EVENT-ACTION         PIC X COMP-X.
```

You can find a copy of this format in the COPY library “crtvars.def”. You may name the data items in the EVENT-STATUS declaration arbitrarily, but the data types, storage, and group structure must match the example given. (For compatibility with older source code, the compiler accepts an EVENT-STATUS item that does not have EVENT-ACTION. The runtime behaves as if EVENT-ACTION contains the value “0”, indicating normal event handling.) The SIGNED-LONG data item, EVENT-DATA-2, may be compiled with any “-Dw” setting (“-Dw” limits the word-size of the target machine). If you use “-Dw16” or “-Dw32”, then you should not run the generated object on a 64-bit machine.

16. *Alphabet-name* is a user-defined word that defines an alphabet name.

17. The optional word “ALPHABET” is required in an alphabet declaration if it immediately follows a SYMBOLIC CHARACTERS declaration.

18. *Literal-1, literal-2,* and *literal-3* may be any literal, but if they are numeric, they must be in the range of 1 through 256.

19. *Literal-2* and *literal-3* must have a size of one character if they are alphanumeric literals.

20. *Literal-1* must also have a size of one character if it is associated with a THROUGH or ALSO phrase.

General Rules

1. The *switch-name* clause associates status names (*cond-name*) and switch names (*mnemonic-name*) with a particular program switch. These can be used to test the on/off status of a switch or to change the switch’s status.

2. The *system-name* clause associates a user-defined *mnemonic-name* with one of the predefined system devices. These names may be used in the ACCEPT and DISPLAY statement to refer to the following devices:

System Name	ACCEPT	DISPLAY
CONSOLE	system input	system output

System Name	ACCEPT	DISPLAY
SYSIN	system input	(illegal)
SYSIPT	system input	(illegal)
SYSOUT	(illegal)	system output
SYSLST	(illegal)	system output
SYSLIST	(illegal)	system output
SYSOUT-FLUSH	(illegal)	system output
SYSERR	(illegal)	error output

The "system input" and "system output" devices are normally the console's keyboard and screen, but may be redirected with operating system commands or with the "-i" and "-o" runtime options. The "error output" device is normally the console screen, but may be redirected with the "-e" runtime option.

3. The SYMBOLIC CHARACTERS clause defines symbolic characters. A symbolic character is a user-defined figurative constant and can be used anywhere a figurative constant may be. For each *name*, the character it represents is set to the character whose ordinal position in the native character set is specified by the corresponding *number*. Note that the ordinal position of a character is one greater than its internal representation. Thus a carriage-return character in ASCII (internal value of "13") would be specified as ordinal position "14".

4. The CLASS clause defines a class name. A class name is used in a class test to determine whether or not a data item entirely consists of a certain set of characters.

   The class name must be unique in the source context. Make sure that there is no name clash between the class name and reserved COBOL words, user-defined variables, enumerators, events, or methods. This is particularly important for COM objects, ActiveX controls, and .NET assemblies, which may have enumerators that use the same identifier as a class name they define. If there is a name clash, rename the class, or if possible, the other element to make the class name unique. Otherwise compiler errors may result. We suggest prepending ActiveX and .NET class names with an "@" sign to avoid ambiguities.

For each numeric literal specified in the CLASS clause, the value of the literal specifies the ordinal number of the character in the native character set to include in the class. For example, "33" would refer to the space character (decimal value 32) in the ASCII character set.

For each alphanumeric literal, the value of the character or characters in the literal specifies the characters to include in the class.

If the THROUGH phrase is used, then *lit-1* and *lit-2* must be numeric literals or alphanumeric literals containing only one character. The set of contiguous characters between *lit-1* and *lit-2* (inclusive) is included in the class. The two literals may be specified in either ascending or descending order.

5. The CURRENCY SIGN clause specifies the PICTURE clause currency symbol. It can be any character from the computer's character set.

6. If no CURRENCY SIGN clause is present, the dollar sign is used.

7. The DECIMAL-POINT IS COMMA clause exchanges the functions of the comma and the period in PICTURE clauses and numeric literals.

8. The NUMERIC SIGN clause specifies that all USAGE DISPLAY numeric data items in the program that do not have an explicit SIGN clause should be treated as if they had a SIGN TRAILING SEPARATE clause. A compile-time option exists to do the same thing. This is usually done to match the behavior of other COBOL systems.

9. The CONSOLE IS CRT clause causes the compiler to assume that:

   - "UPON CRT" is specified for every DISPLAY statement that does not have an UPON phrase.

   - "FROM CRT" is specified for every ACCEPT statement that does not have a FROM phrase.

   This causes DISPLAY and ACCEPT statements without explicit UPON or FROM phrases to interact with the ACUCOBOL-GT Window Manager instead of with the low-level (ANSI-style) console driver. Note

that CONSOLE IS CRT is automatically implied by the compiler unless the "-Ca" compiler option is used. If CONSOLE IS CRT is present in a program compiled with "-Ca", CONSOLE IS CRT takes precedence.

10. The CURSOR clause specifies the name of a data item that will be used to control the console's cursor position throughout the program. If the *cursor-name* data item is four characters long, then the first two characters are the cursor's line number, and the last two characters are the column number. If *cursor-name* is six characters long, then the row and column numbers are each three characters long.

    At the beginning of each ACCEPT statement, *cursor-name* should specify a location in one of the fields being entered. If it does, then the cursor will begin the ACCEPT statement at that location. If *cursor-name* does not specify a valid position, and more than one field is being entered by the ACCEPT statement, the cursor will start at the beginning of the next valid field (the next field described in the Screen Section definition of the *screen-name* referenced in this ACCEPT). If the cursor position is past the last field, or before the first field, then the cursor will be placed at the beginning of the first field.

    At the conclusion of the ACCEPT statement, the cursor's final location will be placed in *cursor-name*.

Note: The ACUCOBOL-GT runtime system will limit the placement of the cursor in some circumstances. It will not place a cursor past the end of valid data in a field. In this case, the cursor will be placed as close to the requested position as possible. Also note that if you specify a CURSOR clause in SPECIAL-NAMES, then you must initialize the cursor location prior to *every* ACCEPT statement.

If you specify a CURSOR clause in SPECIAL-NAMES, you may not use the CURSOR phrase of the ACCEPT statement, because this would lead to multiple specification of the initial cursor position.

If you don't initialize cursor-name with numeric data, you'll receive a "nonnumeric data" warning message on your first ACCEPT statement.

The cursor may not start over a prompt character (except at the beginning of a field), unless that prompt character is a space.

The following code demonstrates how a CURSOR clause could be used with a Format 1 ACCEPT statement:

```
identification division.
program-id. testcurs.
***  This program shows how the CURSOR clause
***  can be used with a Format 1 ACCEPT statement.
environment division.
configuration section.
special-names.
    cursor is cursor-name.
data division.
working-storage section.

01  cursor-name.
    05  c-line      pic 9(3).
    05  c-col       pic 9(3).

01  accept-field    pic x(10) value "abcdefghij".
procedure division.
main-logic.

display window erase.

display accept-field line 6 col 6.

***  Position the cursor over the "c" in the value
***  of accept-field, which is the eighth column
***  on the screen.

   move 6 to c-line.
   move 8 to c-col.
   accept accept-field line 6 col 6 update.
   stop run.
```

11. The CRT STATUS phrase provides a method for returning the termination status of every ACCEPT statement. At the end of each ACCEPT statement, *status-name* is filled in with information regarding how the ACCEPT terminated. Two forms of status values are supported. One form is a three-character group item where each character is treated as a “key” that contains various information. The second form is a simple numeric value where each status condition is identified by a unique numeric value. The next two rules describe these two forms.

12. If *status-name* is a numeric data item, then a unique value will be moved to this item at the end of each ACCEPT statement. This value is the same as the CONTROL KEY value described in the ACCEPT statement. For details on which values are returned, see the discussion of the CONTROL KEY phrase in **section 6.6** of this manual.

13. If *status-name* is a group item, then the first two characters are assigned values according to the following table:

Key 1	**Key 2**	**Meaning**
'0'	'0'	Termination key pressed
'0'	'1'	Auto-skip out of last field
'1'	x'00' - x'FF'	Exception key pressed
'2'	x'00'	End-of-file key pressed
'3'	x'00'	Statement timed out
'9'	x'00'	No items fall within screen

If Key 1 is "1", then Key 2 contains the exception key value of the key that was pressed. For a table of key values, see the heading "CONTROL KEY Phrase" under the **ACCEPT Statement** in **section 6.6**.

The third character always contains the same value that is returned by the CONTROL KEY phrase of the ACCEPT statement, if this value is in the range of 0 to 255. This is the same value returned when *status-name* refers to a numeric data item instead of a group item (rule 12 above).

When Key 1 is set to "0", a normal termination has occurred. Any other value indicates an exception condition.

Note: This form of the CRT STATUS phrase is provided for compatibility with the X/Open COBOL specification as well as for compatibility with some other COBOL systems. This method of determining the status of an ACCEPT statement differs in several details from the other methods of determining the ACCEPT status supported by ACUCOBOL-GT. In particular, the other methods return a single numeric value to describe the status (same as the value stored

in Key 3). We recommend that only one technique be used to test for the ACCEPT status to avoid confusion. The following table summarizes the various statements and phrases available to return an ACCEPT statement's status.

Status Statement	Status Type
CRT STATUS group-item	3-byte key
CRT STATUS numeric-item	Numeric value
CONTROL KEY IS item	Numeric value
ACCEPT item FROM ESCAPE	Numeric value
ON EXCEPTION item	Numeric value

Also note that the formation of Key 2 is tricky. In the case of a normal termination, Key 2 will contain a normal COBOL character digit, but in the other cases, it will contain a binary value. The easiest way to test the status value is to use hexadecimal constants to express the binary value. Alternately, you can declare Key 2 to be COMP-X and test the exception values against numeric literals. Note that some other COBOL systems define Key 2 to be "PIC 99 COMP". If you convert programs that use this construct, be sure to use the "-D1" or "-Dm" compile-time flags to cause this data item to be stored in one character. If you do not do this, then Key 2 will occupy two characters and return invalid values.

14. The ALPHABET clause specifies the alphabet to be used for character translations and collating sequences.

    An alphabet may be used in the following circumstances:

    a. In the CODE-SET phrase of a sequential file's FD. The alphabet specifies a character translation map. A Format 2 alphabet may not be used for this purpose.

    b. In the COLLATING SEQUENCE phrase of an indexed file's SELECT. This specifies an alternate collating sequence for the file's keys.

    c. In the COLLATING SEQUENCE phrase of a SORT or MERGE statement. Here the alphabet specifies the collating sequence to use for key comparisons.

d. In the PROGRAM SEQUENCE phrase of the OBJECT-COMPUTER paragraph. This specifies the collating sequence for any alphanumeric comparisons done in the program. It also specifies the default collating sequence for SORT and MERGE verbs. The character that is first in the program collating sequence is treated as the LOW-VALUES character for the program. The character that is last in the program collating sequence is treated as the HIGH-VALUES character for the program. The one exception to this is that in Special-Names, LOW-VALUES and HIGH-VALUES always refer to the first and last characters in the native collating sequence.

e. In a SYMBOLIC CHARACTERS clause in SPECIAL-NAMES, to indicate the alphabet to which the symbolic character belongs.

A Format 2 alphabet is used to describe a collating sequence. Explicitly named characters are listed in the order of their positions in the new collating sequence.

Any characters in the native collating sequence that are not explicitly named in the ALPHABET clause assume a position greater than any of the explicitly named characters. The relative order of these unnamed characters remains the same as in the native collating sequence.

If a literal in the ALPHABET clause is numeric, it designates a character by specifying that character's ordinal position in the native character set. For example, 66 would designate the letter A in the ASCII character set.

If the literal is alphanumeric, it is the actual character. For alphanumeric literals that contain more than one character, the characters are assigned successive ascending positions in the new collating sequence.

Here's an example :

```
ALPHABET TINY IS "01", "Z", "AB", SPACE
```

This alphabet contains 6 characters. Character 0 is the first character in the sequence, character 1 is the second. Character "Z" is the third, and characters "A" and "B" are the fourth and fifth. The space character is the sixth character. If you were to sort a file using this alphabet, all items that started with "Z" would appear before items that started with "A" or "B". Anything that started with a space would be last.

Any characters in the native collating sequence that are not explicitly named assume a position in the new collating sequence greater than any of the explicitly named characters. The relative order of these characters remains unchanged from the native collating sequence. In the example above, this means that anything that starts with "C" comes after anything that starts with spaces, and anything starting with "D" comes after "C".

If the THROUGH phrase is specified, the set of contiguous characters in the native character set beginning with *literal-1* and ending with *literal-2* are assigned successive ascending positions in the new collating sequence. The THROUGH phrase may specify characters in ascending or descending sequence.

For example, the following alphabet will sort the alphabetic characters backwards:

```
ALPHABET REV-ALPHA IS "Z" THROUGH "A",
        "z" THROUGH "a"
```

If the ALSO phrase is specified, then *literal-1* and *literal-3* are assigned the same position in the collating sequence. This is one of the most useful capabilities of the Format 2 ALPHABET clause. For example, the following alphabet will cause the upper case and lower case of each letter to be treated as the same character for sorting:

```
ALPHABET NO-CASE IS 1 THRU 65,   'A' ALSO 'a',
'B' ALSO 'b',   'C' ALSO 'c',   'D' ALSO 'd',
'E' ALSO 'e',   'F' ALSO 'f',   'G' ALSO 'g',
'H' ALSO 'h',   'I' ALSO 'i',   'J' ALSO 'j',
'K' ALSO 'k',   'L' ALSO 'l',   'M' ALSO 'm',
'N' ALSO 'n',   'O' ALSO 'o',   'P' ALSO 'p',
'Q' ALSO 'q',   'R' ALSO 'r',   'S' ALSO 's',
'T' ALSO 't',   'U' ALSO 'u',   'V' ALSO 'v',
'W' ALSO 'w',   'X' ALSO 'x',   'Y' ALSO 'y',
'Z' ALSO 'z'
```

The "1 THRU 65" phrase causes the portion of ASCII that exists in front of "A" to be sorted in its normal sequence ("A" is the 66th character in ASCII). Then each lower-case character is mapped to its corresponding upper-case character. The remaining characters then follow implicitly.

15. The following IBM DOS/VS COBOL system names are supported by ACUCOBOL-GT in the SPECIAL-NAMES paragraph if the "-Cv" compiler option is used:

```
SYSPCH
SYSPUNCH
C01 through C12
CSP
S01 through S05
```

To enable printing to printer channels C01-C12, set the runtime configuration variable called "**COBLPFORM**" as described in Appendix H in the *ACUCOBOL-GT Appendices* manual.

See Chapter 5, "IBM DOS/VS COBOL Conversions," in *Transitioning to ACUCOBOL-GT* for more information.

16. The following HP COBOL special names are supported by ACUCOBOL-GT in the SPECIAL-NAMES paragraph if the "-Cp" compiler option is used:

```
NO SPACE CONTROL
TOP
```

See Chapter 4, "HP COBOL Conversions," in *Transitioning to ACUCOBOL-GT* for more information.

General Rules - Screen Control Entry

In the following rules, the four elementary items belonging to the SCREEN CONTROL group item are referenced by the names in this example:

```
01  SCREEN-CONTROL.
    03  ACCEPT-CONTROL     PIC 9.
    03  CONTROL-VALUE      PIC 999.
    03  CONTROL-HANDLE     USAGE HANDLE.
    03  CONTROL-ID         PIC X(2) COMP-X.
```

1. The following statement allows an embedded procedure to control its ACCEPT statement:

```
[ SCREEN CONTROL IS control-name ]
```

For more information, see **Section 5.9.6, "PROCEDURE Clause,"** and Book 1, **Section 6.5.5**, "Using Screen Section Embedded Procedures."

2. Input and update fields in a Screen Section entry are given *field numbers.** The compiler computes the field number for any Screen Section entry by examining all of the input and update fields in that

entry's level 01 group item. Each input and update field in a level 01 Screen Section entry is numbered sequentially, starting at one. For example, consider the following Screen Section entry:

```
01  SCREEN-1.
    03  LITERAL-1 VALUE "Field 1: ".
    03  FIELD-1, PIC X(5) TO WS-1.
    03  LITERAL-2 VALUE "Some data: ", LINE + 1.
    03  DATA-1, PIC X(5) FROM WS-2.
    03  LITERAL-3 VALUE "Field 2: "., LINE + 1.
    03  FIELD-2, PIC X(5) USING WS-3.
```

This Screen Section entry has two input or update fields: FIELD-1 and FIELD-2. In this case, FIELD-1 is field number 1 and FIELD-2 is field number 2. Note that LITERAL-1, LITERAL-2, and DATA-1 do not receive field numbers because they do not contain a TO or USING phrase—DATA-1 is a display-only (FROM) field. The literals prompt the end user for entries.

*Graphical controls in the Screen Section are also assigned field numbers. The rules that govern how field numbers are assigned to graphical controls are given in general rule 6 of the Format 2 Screen Description, in section 5.9, "Screen Description Entry."

3. Prior to executing an embedded procedure (see **section 5.9.6**), an ACCEPT statement initializes the SCREEN CONTROL variable. It sets ACCEPT-CONTROL depending on the reason for entry (if it is a *notify* ("NTF-…") event, ACCEPT-CONTROL is set to "1"; otherwise, the default is "0"), and it sets CONTROL-VALUE to the field number of the Screen Section entry that is executing the embedded procedure.

   On entry to an embedded procedure, CONTROL-HANDLE contains the handle of the current control, and the CONTROL-ID field contains its ID. If the current Screen Section item (the one that names the embedded procedure) is not a graphical control, CONTROL-HANDLE and CONTROL-ID are set to NULL and "0", respectively.

   When the ACCEPT statement terminates, it sets ACCEPT-CONTROL to 0 and sets CONTROL-VALUE to the field number of the last field to have the cursor. This will be zero if the Screen Section entry contains no fields. CONTROL-HANDLE and CONTROL-ID fields contain the

handle and ID of the graphical control that was active when the ACCEPT terminated. If the ACCEPT terminated while in a textual (i.e., non-graphical) field, they are set to NULL and "0", respectively.

4. When an after procedure or exception procedure returns control to its ACCEPT statement, the value of SCREEN CONTROL determines what happens next. By setting this value in your after or exception procedure, you can cause the program to skip fields, continue ACCEPTING data, or terminate the ACCEPT with or without an exception. ACCEPT-CONTROL serves as a flag that is checked to determine how to proceed; SCREEN-CONTROL provides a needed value, as shown in this list:

   a. If ACCEPT-CONTROL is 0, the ACCEPT statement continues normally.

   b. If ACCEPT-CONTROL is 1, the ACCEPT statement moves the cursor to the field identified by the value of CONTROL-VALUE. The ACCEPT statement then continues from there.

   c. If ACCEPT-CONTROL is 2, the ACCEPT statement terminates normally. The value of CONTROL-VALUE determines the termination value of the ACCEPT statement. You can determine its value by examining CRT STATUS or by using the ACCEPT FROM ESCAPE KEY verb.

   d. If ACCEPT-CONTROL is 3, the ACCEPT statement terminates with an exception, assuming that exceptions are allowed. The value of CONTROL-VALUE sets the exception value of the ACCEPT statement. You may use CRT STATUS or the ACCEPT FROM ESCAPE KEY verb to determine the statement's exception value.

   e. If ACCEPT-CONTROL is 4, control is transferred to the graphical control whose ID matches CONTROL-ID. This works identically to setting ACCEPT-CONTROL to "1", except that the CONTROL-ID field is used and the search is made using the control's ID instead of the field numbers.

   f. If none of the preceding applies, the ACCEPT statement continues normally.

5. If you set ACCEPT-CONTROL to 1, several special cases exist:

a. If you set CONTROL-VALUE to zero, the ACCEPT statement will remain in the current field.

b. If you set CONTROL-VALUE to a field number that does not exist, the ACCEPT statement will terminate. In this case, the CRT STATUS value for the ACCEPT statement will be zero for a numeric CRT STATUS or “0”, “2”, x“00” for a group-item CRT STATUS.

c. If you set CONTROL-VALUE to the field number of a protected field, control will pass to the first unprotected field with a higher field number. If no such field exists, the ACCEPT statement will terminate as in case (b) above.

6. When a Screen Section ACCEPT statement executes, it examines the value of SCREEN CONTROL. If the ACCEPT-CONTROL field is 1, then the ACCEPT statement starts at the field identified by CONTROL-VALUE. This overrides any initial field identified by the CURSOR Special-Names entry. Note that this is usually easier than using the CURSOR clause to identify a starting point in a Screen Section ACCEPT. If the specified field does not exist (or is protected) the cursor is placed at the numerically closest legal field. If two fields are equally close, the one with the larger field number is used.

General Rules - Event Status Entry

EVENT-STATUS is used to identify which data item is to receive information about screen events.

In the description below, the seven elementary items belonging to the EVENT-STATUS group item are referenced by the names in this example:

```
01 EVENT-STATUS.
   03  EVENT-TYPE           PIC X(4) COMP-X.
   03  EVENT-WINDOW-HANDLE  USAGE HANDLE OF WINDOW.
   03  EVENT-CONTROL-HANDLE USAGE HANDLE.
   03  EVENT-CONTROL-ID     PIC X(2) COMP-X.
   03  EVENT-DATA-1         USAGE SIGNED-SHORT.
   03  EVENT-DATA-2         USAGE SIGNED-LONG.
   03  EVENT-ACTION         PIC X COMP-X.
```

When a system event occurs during an ACCEPT statement, the EVENT-STATUS data item is filled with the following information:

EVENT-TYPE

Holds a value that uniquely identifies the kind of event that occurred. The valid types are described in Chapter 6 of Book 2, *User Interface Programming*.

EVENT-WINDOW-HANDLE

Holds the handle of the floating window in which the event occurred. If the event occurred in a control, this will be the handle of the floating window that contains the control.

EVENT-CONTROL-HANDLE

Holds the handle of the control in which the event occurred. If the event did not occur in a control, this item is set to NULL.

EVENT-CONTROL-ID

Holds the ID of the control in which the event occurred. IDs are assigned by the application when each control is created. If the event did not occur in a control, this item will have the value zero.

EVENT-DATA-1

Holds information about the event that is unique for each EVENT-TYPE. For many events, this value will always be zero.

EVENT-DATA-2

Also holds information about the event that is unique for each EVENT-TYPE. For many events, this value will always be zero.

EVENT-ACTION

Holds a value that determines the continued handling of an event when an event procedure terminates. On entry to the procedure, EVENT-ACTION is set to zero. The following values are meaningful on exit from the procedure (symbolic names in "acugui.def"):

EVENT-ACTION-NORMAL	(value 0) The event is processed normally, causing the control to terminate for terminating events.
EVENT-ACTION-TERMINATE	(value 1) The event is processed normally, and then it terminates the active control. This action forces termination of events that do not normally terminate.
EVENT-ACTION-CONTINUE	(value 2) The event is processed normally, but it does not terminate the active control, even if it would ordinarily do so.
EVENT-ACTION-IGNORE	(value 3) The event is not processed further, but it does not terminate the active control. We do not recommend this action because it short-circuits the runtime's event handler. Events receive a certain amount of processing before the event procedure is entered. Ignoring an event does not prevent this processing from occurring.

EVENT-ACTION-FAIL	(value 4) This setting is used in response to certain events to indicate that a specific action should be taken, usually to prevent the event from taking its normal action. Events that use this setting state that they do so in the event description, along with a description of the effects of setting it.
EVENT-ACTION-FAIL -TERMINATE	(value 7) The effect of this setting is exactly the same as that of EVENT-ACTION-FAIL with the additional effect of EVENT-ACTION-TERMINATE: after performing the "fail" operation, the control terminates with an exception status of W-EVENT.

4.3 Input-Output Section

The INPUT-OUTPUT Section describes the I/O environment that the program will be using. The header for this section is optional.

4.3.1 File-Control Paragraph

The FILE-CONTROL paragraph contains descriptions of the physical aspects of the files the program uses.

General Format

```
[ FILE-CONTROL. ]  { file-control-entry } ...
```

or

```
{ file-control-entry } ...
```

File-control-entry has one of the following formats:

Format 1 - Sequential Files

```
SELECT  [ OPTIONAL ] file-name

   ASSIGN TO [ DYNAMIC ] [ device  ] [ file-spec ]
             [ EXTERNAL]

 [ [ ORGANIZATION IS ] [BINARY] SEQUENTIAL ]
                       [RECORD]
                       [LINE  ]

 [ ACCESS MODE IS SEQUENTIAL ]

 [ RESERVE {number} [ALTERNATE] [AREA ]]
           {NO    }             [AREAS]

 [ LOCK MODE IS {EXCLUSIVE}
                {AUTOMATIC}
                {MANUAL   }

 [ RECORD DELIMITER IS [ STANDARD-1        ]
 [ FILE STATUS IS status-variable [status-variable-2]]

 [ PADDING CHARACTER IS pad-char ] .
```

Format 2 - Relative Files

```
SELECT  [ OPTIONAL ] file-name

   ASSIGN TO [ DYNAMIC ] [ device  ] [ file-spec ]
             [ EXTERNAL]

 [ ORGANIZATION IS ] RELATIVE

 [ ACCESS MODE IS

   { SEQUENTIAL [ RELATIVE KEY IS rel-key ] } ]
   { RANDOM       [RELATIVE] KEY IS rel-key   }
                  [ACTUAL]
   { DYNAMIC      RELATIVE KEY IS rel-key   }

 [ LOCK MODE IS { EXCLUSIVE                    }
                { AUTOMATIC [ multiple-option ] }
```

```
                { MANUAL    [ multiple-option ] }

 [ RESERVE {number} [ALTERNATE] [AREA ]]
           {NO    }             [AREAS]

 [ FILE STATUS IS status-variable [status-variable-2] ] .
```

See the multiple-option format at the end of the formats.

Format 3 - Indexed Files

```
SELECT  [ OPTIONAL ] file-name

   ASSIGN TO [ DYNAMIC ] [ device  ] [ file-spec ]
             [ EXTERNAL]

 [ WITH  {COMPRESSION} ... ]
         {ENCRYPTION }

 [ COMPRESSION CONTROL VALUE IS factor ]

 [ ORGANIZATION IS ] INDEXED

 [ ACCESS MODE IS  {SEQUENTIAL} ]
                   {RANDOM    }
                   {DYNAMIC   }

 [ RECORD KEY IS key-name [= seg-name ...]
    [ WITH [NO] DUPLICATES ] ] ...

 [ ALTERNATE RECORD KEY IS alt-name [= seg-name ...]
    [ WITH [NO] DUPLICATES ] ] ...

 [ LOCK MODE IS { EXCLUSIVE [ WITH MASS-UPDATE ] }
                { AUTOMATIC [ multiple-option  ] }
                { MANUAL    [ multiple-option  ] }

 [ RESERVE {number} [ALTERNATE]  [AREA ]]
           {NO    }              [AREAS]

 [ FILE STATUS IS status-variable [status-variable-2] ]

 [ COLLATING SEQUENCE IS alphabet-name ] .
```

Format 4 - Sort Files

```
SELECT file-name

   ASSIGN TO [ DYNAMIC ] [ device  ] [ file-spec ]
             [ EXTERNAL]

 [ FILE STATUS IS status-variable [status-variable-2]] .
```

Note: *multiple-option* has the following format for both Format 2 and Format 3.

```
WITH { { LOCK ON } [ MULTIPLE ] { RECORD  } } [WITH ROLLBACK]

     { ROLLBACK                          }
```

Syntax Rules

1. *File-spec* must be either a nonnumeric literal or the name of an alphanumeric Working-Storage data item. See Book 1, section 6.1.3, for information about creating Vision indexed files.

2. *Device* must be one of these words: INPUT, OUTPUT, INPUT-OUTPUT, RANDOM, DISK, DISC, PRINT, PRINTER, PRINTER-1, TAPE, CASSETTE, CARD-PUNCH, CARD-READER, CONSOLE, MAGNETIC-TAPE, DISPLAY, KEYBOARD, SORT, MERGE, SORT-MERGE or SORT-WORK. The last four may be used only with sort files.

3. If DYNAMIC is specified, *file-spec* must be specified and must be the name of a data item. This data item need not be defined in the program, although it can be.

4. If EXTERNAL is specified, *file-spec* must be specified and must be a user-defined COBOL word.

5. Status-variable must be the name of an alphanumeric (or USAGE DISPLAY numeric) Working-Storage or Linkage data item with a size of 2 characters. Status-variable-2 must be the name of a group item that is 6 characters (this is not checked by the compiler).

6. *Rel-key* must name an unsigned integer data item. It must not be in the record description entry for the same file.

7. *Factor* must be a numeric literal from zero to 100, inclusive.

8. The key of an indexed file may have any PICTURE and USAGE. Regardless of the PICTURE and USAGE specified, the key is always treated as an alphanumeric data item when the sort order of the file is determined (the individual bytes are compared with the collating sequence).

9. *Seg-name* must name a data item in the same file's record description entry. A seg-name may not be a group item that contains variable-occurrence data items.

10. If *seg-name* is used to define a split RECORD KEY, then *key-name* is a user-defined word. Otherwise, *key-name* must name a data item in the same file's record description entry. It may not be a group item that contains variable-occurrence data items.

11. If *seg-name* is used to define a split ALTERNATE RECORD KEY, then *alt-name* is a user-defined word. Otherwise, *alt-name* must name a data item in the same file's record description entry. It may not be a group item that contains variable-occurrence data items.

12. *Number* must be an integer literal.

13. *Alphabet-name* is the name of an alphabet declared in the Special-Names paragraph.

14. *Pad-char* must be either a single-character literal or a single-character alphanumeric data item. When a PADDING CHARACTER is specified, the last block of the file is padded with *pad-char*. When the file is read, any final portion of a block that consists solely of padding characters is skipped.

15. SELECT must be the first clause in a FILE-CONTROL entry. The other clauses may follow in any order. The SELECT clause may appear in Area A; all other clauses must appear in Area B.

16. Each file described in the Data Division must be specified exactly once in the FILE-CONTROL paragraph.

17. Each file described by a SELECT clause must have exactly one corresponding file description in the Data Division.

18. ORGANIZATION IS RECORD SEQUENTIAL is synonymous with ORGANIZATION IS BINARY SEQUENTIAL.

General Rules

1. When the FILE STATUS clause is specified, a value will be moved into status-variable after the execution of every statement that references the corresponding file. This value indicates the status of the statement. (See **section 6.4.7, "I/O Status."**) Status-variable-2 is treated as commentary by the compiler.

2. The ACCESS MODE clause specifies the order in which records are read or written. If it is not specified, SEQUENTIAL is implied.

3. For sequential access, the records are accessed according to the organization of the file:

   - Sequential files - The sequence is the same as that established by the execution of WRITE statements that created the file.

   - Relative files - The sequence is the order of ascending relative record numbers for the file's existing records.

   - Indexed files - The sequence is the order of ascending record key values for the file's current key of reference.

4. Random access indicates that the file will be accessed only by key value.

5. Dynamic access indicates that the file will be accessed both randomly and sequentially.

6. RELATIVE KEY results in record key numbers that are one-based. ACTUAL KEY may be specified for RANDOM access mode and when compiling version 8.1 and greater objects in IBM DOS/VS ("-cv") and HP3000 ("-cp") compatibility modes. For such files, the record key numbers will be zero-based. For example, if you have a relative file with a fixed record length of 3 bytes and a relative file with the following contents:

   ```
   AAABBBCCC
   ```

   the record keys for the different modes are:

   ```
        RELATIVE  ACTUAL
   AAA:    1         0
   BBB:    2         1
   CCC:    3         2
   ```

7. The ASSIGN clause specifies the association of the file to a storage device. The rules for interpreting the ASSIGN clause are described in Book 1, section 2.8. If the *file-spec* phrase is missing, then *file-name* will be treated as an alphanumeric literal and substituted for it (exception: rule 8.a below). In this case, *file-name* should conform to the host operating system's rules for file names. Note that the ASSIGN clause is required even if both *device* and *file-spec* are missing.

8. The *device* phrase of the ASSIGN clause is not required. If it is specified, it can affect the processing of the file in a variety of ways. If the file is not a sequential file, then the *device* phrase is ignored. If it is a sequential file, then the following applies depending on the *device* phrase used:

    a. **PRINT, PRINTER, PRINTER-1** - A sequential file marked with one of these *device* phrases will be treated as a "print" file. Print files may not be opened for INPUT or I/O. When records are written to a print file, trailing spaces are first removed from the record. Print files have printer carriage control information added to them as specified by the WRITE statements that add records to the file. If "PRINTER" or "PRINTER-1" is specified, and no *file-spec* is specified, then the external file name is treated as "PRINTER" or "PRINTER-1" (unless RM/COBOL compatibility mode is being used, in which case rule 7 applies instead). Normally, these names are translated at runtime to the name of the system spooler. This is an exception to rule 7 above.

    b. **CARD-PUNCH, CARD-READER, CASSETTE, INPUT, INPUT-OUTPUT, MAGNETIC-TAPE, OUTPUT** - Any of these *device* phrases indicates that trailing spaces should be removed from records before they are added to the file. This will have effect only if the file is a "line" sequential file. When records are read from one of these files, the records are automatically padded with spaces to reach the maximum record size. A file with one of these designators may not be opened for I/O.

    c. **DISPLAY, KEYBOARD** - Causes the default file type to be "line" sequential. You may override this by specifying the file type explicitly in the ORGANIZATION clause. This rule is provided for compatibility with ICOBOL, which uses this method for specifying line sequential files.

d. **RANDOM, DISK, DISC, TAPE, CONSOLE** - Indicates no additional processing. This is the same as if the *device* phrase were omitted.

e. **MERGE, SORT, SORT-MERGE, SORT-WORK** - Also indicates no additional processing. These *device* phrases may be associated with a sort file only.

9. The word DYNAMIC in an ASSIGN phrase indicates that *file-spec* is the name of a variable that contains the file's name. Because this is the normal meaning of *file-spec* when it refers to a variable, the word DYNAMIC is largely commentary.

10. When DYNAMIC is specified, if *file-spec* refers to a variable that is not otherwise defined, the compiler creates a Working-Storage variable by that name that is PIC X(256). It is the program's responsibility to move a valid file name to this data item prior to opening the file.

11. The word EXTERNAL in an ASSIGN phrase indicates that the COBOL word that makes up *file-spec* is the name of the file itself. This name is processed first by ignoring any characters that appear before the last hyphen in the word (including the hyphen itself). For example:

```
ASSIGN TO EXTERNAL UT-S-MYFILE
```

results in "MYFILE" being used for the file name. In other COBOL systems, this name is normally assigned to a specific file name using environment variables. This kind of name mapping occurs automatically under ACUCOBOL-GT. There is no special meaning associated with ASSIGN phrases containing the EXTERNAL option. Such files have name mapping applied through the environment just like all other files.

Note: If neither DYNAMIC nor EXTERNAL is included in the ASSIGN clause, you can use the "--fileAssign=" compiler option to specify DYNAMIC or EXTERNAL at compile time. See **Section 2.2.7**, in Book 1, *ACUCOBOL-GT User's Guide*.

---

12. The WITH COMPRESSION phrase of the ASSIGN clause specifies that file record compression is desired. This phrase must be specified before the ORGANIZATION IS INDEXED phrase. The Vision file

system supports compression, but not all file systems do. The WITH COMPRESSION phrase takes effect only when the file is initially created or re-created via the OPEN statement. When no compression *factor* is specified (see next paragraph), WITH COMPRESSION uses the default compression factor (70).

A compression *factor* other than the default may be selected via the COMPRESSION CONTROL VALUE IS clause. The *factor* must be a numeric literal from zero (meaning no compression) to 100 (maximum compression). A compression factor of 1 is equivalent to the default compression.

The exact meaning of the compression factor depends upon the host file system. See Book 1, section 6.1.3, for specifics about the Vision file system.

13. The WITH ENCRYPTION phrase specifies that record encryption is desired on the file. Encryption is currently available with the Vision indexed file system only. The ENCRYPTION clause takes effect only when the file is initially created or re-created via the OPEN statement.

14. The ORGANIZATION clause specifies the logical structure of the file. This is established when a file is first created and may not be changed. If it is absent, then SEQUENTIAL organization is implied. Records stored in an ORGANIZATION IS RELATIVE file are uniquely identified by record number. The relative record number of a given record specifies the record's ordinal position in the file. The first record has a relative record number of one.

15. Records in an ORGANIZATION IS INDEXED file are uniquely identified by the values in the record's primary key.

16. The primary key is identified by the RECORD KEY clause. Records are ordered in ascending collating sequence by the primary key. If the WITH DUPLICATES phrase is present, the primary key may contain duplicate values, if the indexed file system supports them. Vision supports duplicate primary key values. If WITH DUPLICATES is used with a file system that does not support them, when the file is created via the OPEN statement a status of "0M" is returned, indicating that the file was successfully created but that duplicate primary keys are not supported. When WITH DUPLICATES is used with Vision

and other file systems that support it, the rules that govern how REWRITE and DELETE operations are handled are determined by the file system. The rules for Vision are as follows:

a. If the last record locked via a READ statement is still locked and it matches the primary key value specified in a REWRITE or DELETE statement, that record is the record rewritten or deleted.

b. Otherwise, the first record with the matching key value is rewritten or deleted.

For information about how HP e3000 KSAM handles REWRITE and DELETE with duplicate primary keys, see Chapter 4, "HP COBOL Conversions," in *Transitioning to ACUCOBOL-GT.*

The WITH NO DUPLICATES phrase is commentary. By default, duplicate primary key values are not allowed.

17. The ALTERNATE RECORD KEY clause specifies additional record keys for an indexed file. If the WITH DUPLICATES phrase is present, then these key values may contain duplicated values. Otherwise, each key value must be unique for a given key.

    You may specify the word "NO" in front of the word "DUPLICATES" in a declaration of an alternate indexed file key. This is useful for ICOBOL compatibility mode where, by default, alternate keys allow duplicates.

18. Up to 16 seg-names may be specified in Vision Version 4 to indicate that a primary or alternate key consists of non-contiguous data elements. In Vision Version 3, up to six seg-names may be specified, and in Version 2, only one seg-name may be specified. The key-name or alt-name is then a user-defined word that can be used in READ and START.

19. The OPTIONAL phrase, if specified, indicates that the file need not be present when the program is run. The exact effects of this phrase are detailed in the discussion of the **OPEN Statement**. Note that the "-Fp" compile option causes all files to be treated as if the OPTIONAL phrase is present.

20. The LOCK MODE clause specifies how file and record locking should be handled for the file. Each mode has the following characteristics:

AUTOMATIC Each time a record is read from a file open for I/O, that record is locked unless the WITH NO LOCK option is used on the READ statement. Files open for INPUT do not lock records.

MANUAL Records read from a file with manual locking are locked only if the WITH LOCK option is used on the READ statement. Like automatic mode, files open for INPUT do not lock records even if WITH LOCK is specified.

EXCLUSIVE Exclusive mode files are opened with a lock on the entire file. No locking options may be specified on an OPEN statement associated with an exclusive mode file. Instead, files opened for INPUT are treated as if they were opened with the ALLOWING READERS phrase, and files opened for OUTPUT, I-O, or EXTEND are treated as if they were opened with the ALLOWING NO OTHERS phrase. If the WITH MASS-UPDATE phrase is used, then the MASS-UPDATE option is implied for each OPEN (except for OPEN INPUT). See the **OPEN Statement** for details on these options.

21. If the COLLATING SEQUENCE phrase is used, the *alphabet-name* is the name of an alphabet declared in Special-Names. This alphabet can be standard or can be a custom alphabet defined by the programmer to allow special handling. For example, upper-case and lower-case letters could be mapped together so that two keys that are alphabetically the same (but differ in case) would be treated as the same letter. European character sets can also be re-ordered in Special-Names (so that keys are sorted alphabetically).

22. If the LOCK ON MULTIPLE RECORDS phrase is used, then the program may lock more than one record in the file at once. If the MULTIPLE option is not used, then each I-O statement automatically unlocks the currently locked record before executing. When the MULTIPLE option is used, then record locks are released only when an UNLOCK or a CLOSE statement is executed for the file. The ROLLBACK clause is useful when you compile with “-Fl”, which

enables single locking rules as the lock mode default. When the ROLLBACK clause is used with this phrase, multiple locking rules are enabled for the file, regardless of the compiler option used.

23. If the LOCK phrase is omitted, LOCK MODE IS AUTOMATIC is implied, unless the "-Fm" compiler option is used, in which case LOCK MODE IS MANUAL is implied. In ICOBOL compatibility mode ("-Ci"), the default is LOCK MODE IS MANUAL WITH MULTIPLE RECORDS.

24. The RECORD DELIMITER and RESERVE AREA clauses are treated as commentary by the compiler.The RESERVE AREA clause is treated as commentary by the compiler.

25. If the ROLLBACK clause is specified, then WITH LOCK ON MULTIPLE RECORDS will automatically be in effect. However, if you compile with the "-Fl" option, then you must specify multiple locking rules for the files that need them.

26. If the ROLLBACK clause is specified, the runtime will automatically effect a START TRANSACTION before opening the file, and a COMMIT after opening it. Thus, every OPEN of the file will automatically be done within a transaction; the COBOL code need not explicitly include the START TRANSACTION and COMMIT.

27. It is possible that a RECORDING MODE clause may appear in the IBM DOS/VS COBOL "-Cv" compatibility mode and be ignored by the ACUCOBOL-GT compiler. See Chapter 5, "IBM DOS/VS COBOL Conversions," in *Transitioning to ACUCOBOL-GT* for more information.

4.3.2 I-O-Control Paragraph

The I-O-CONTROL paragraph specifies input-output techniques to be used for the program's files.

General Format

```
I-O-CONTROL

 [ APPLY {LOCK-HOLDING } ... ON {file} ... ] ...
```

```
          {PRINT-CONTROL}

[ SAME [RECORD     ] AREA FOR {file} ... ] ...
       [SORT       ]
       [SORT-MERGE]

[ MULTIPLE FILE TAPE CONTAINS

  { tape-file [ POSITION pos ] } ... ] ... .
```

Syntax Rules

1. *File* and *tape-file* name a file described by a SELECT clause in the FILE-CONTROL paragraph.
2. A *file* name cannot appear in more than one SAME RECORD AREA clause.
3. If *file* is a sort file, one of the RECORD, SORT, or SORT-MERGE options must be used in a SAME AREA clause.
4. SORT and SORT-MERGE are equivalent.
5. At least one *file* must be a sort file if the SORT or SORT-MERGE option is used.
6. *Pos* must be an integer literal.
7. You may put APPLY clauses of the IBM DOS/VS COBOL type into the I-O-CONTROL paragraph, but only when the compiler is in the IBM DOS/VS COBOL compatibility mode. The clauses of the I-O-CONTROL paragraph may be arranged in any order, and the existing APPLY clause may be used, regardless of the compiler mode. See Chapter 5, "IBM DOS/VS COBOL Conversions," in *Transitioning to ACUCOBOL-GT* for more information.

General Rules

1. The APPLY clause modifies various characteristics of each *file*.
   a. If the PRINT-CONTROL option is specified, then the named *file* must be a sequential file. This option causes the file to be treated as a print file. This has the same effect as specifying "PRINT" in the *file's* ASSIGN clause.

b. If the LOCK-HOLDING option is used, then record locks on the file will not be automatically released by any I-O statement. Instead, only the UNLOCK and CLOSE statements will release any record locks held on the file. This option has the same effect as the LOCK ON MULTIPLE RECORDS phrase in the *file's* SELECT.

2. The SAME AREA clause indicates that the compiler should share file information areas for the named files. ACUCOBOL-GT automatically applies the most efficient use of memory possible for file information areas and treats this clause as commentary. In ICOBOL compatibility mode, the SAME AREA clause is treated as a SAME RECORD AREA clause (see below).

3. The SAME RECORD AREA clause indicates that the named files should share the same memory area for their current logical records.

4. The SAME RECORD AREA clause is identical to an implicit redefinition of the shared files' record areas.

5. The SAME SORT AREA clause indicates that the same memory should be used for each *file*. Because ACUCOBOL-GT dynamically allocates memory to sort files as needed and then discards the memory when finished, this phrase is treated as commentary by the compiler.

6. The MULTIPLE FILE TAPE clause is treated as commentary.

5 Data Division

Key Topics

5.1 Data Structures

The Data Division describes the data used by the program in both physical and logical terms.

COBOL data structures are defined and described in the following sections.

5.1.1 Record Description

All user-defined data used in a COBOL program belongs to one or more *logical records*. A logical record is defined by a *record description entry.* Logical records may correspond to actual disk records (by being defined in the File Section), or they may simply be areas of computer memory used by the program. A record description entry is itself composed of one or more *data description entries*. Each data description entry defines one COBOL data item.

Logical records can be divided into a hierarchy of individual data items. Subdivision can continue for each of the record's parts. The lowest level subdivision of a record is the *elementary data item*. Elementary data items are never subdivided. A logical record is either an elementary data item or a set of elementary data items.

A *group item* is a piece of data that contains other subordinate data items. These subordinate items may be either elementary data items or other group items. The lowest level group item is always a sequence of one or more elementary data items.

5.1.2 Level-Numbers

Level-numbers are used to describe the hierarchical organization of a record. Level-numbers that describe this hierarchy range from 01 through 49.

The topmost data item is the record. It always has a level-number of 01. Items that are included in the record have greater (although not necessarily consecutive) level-numbers.

All items subordinate to a group item must have level-numbers greater than the group's level-number. The end of a group item is delimited by the next data description entry that has a level-number less than or equal to the group's level-number.

Four special level-numbers are used to specify special types of data. They are never used in a hierarchical structure. Instead, they define the following special types:

- Level-number 66 identifies a RENAMES item that regroups other data items.
- Level-number 77 identifies an elementary data item in the Working-Storage or Linkage sections. These are essentially identical to a level 01 elementary data item. The level-number is used to emphasize that the data item is not part of a hierarchy and cannot itself be subdivided.
- Level-number 78 associates a value with the name of a constant.
- Level-number 88 identifies a condition-name and its values.

The following example shows how level-numbers define a record's hierarchy and shows how records, groups, and elementary items interact. The items are indented to display the hierarchy. This is a recommended programming practice but is not required by COBOL.

```
01  EMPLOYEE-RECORD.                    (record)
    03  EMPLOYEE-KEY.                   (group)
        05  DEPARTMENT-CODE             (elementary)
        05  EMPLOYEE-NUMBER             (elementary)
    03  EMPLOYEE-IDENTIFICATION.        (group)
        05  EMPLOYEE-NAME               (elementary)
        05  EMPLOYEE-ADDRESS.           (group)
            07  STREET-ADDRESS-1        (elementary)
            07  STREET-ADDRESS-2        (elementary)
            07  CITY                    (elementary)
            07  STATE                   (elementary)
            07  ZIP-CODE                (elementary)
        05  EMPLOYEE-RACE               (elementary)
        05  MARRIAGE-STATUS             (elementary)
    03  PAYROLL-INFORMATION.            (group)
        05  SALARY                      (elementary)
        05  PAY-FREQUENCY               (elementary)
        05  DEDUCTION-CODE-1            (elementary)
```

```
05  DEDUCTION-CODE-2            (elementary)
05  SICK-ACCRUAL-RATE           (elementary)
05  VACATION-ACCRUAL-RATE       (elementary)
```

5.1.3 Classes of Data

Depending on how a data item is defined, it belongs to one of the five following *categories*:

1. Alphabetic
2. Alphanumeric
3. Alphanumeric Edited
4. Numeric
5. Numeric Edited

These categories are further grouped into the following *classes*:

1. Alphabetic class: alphabetic
2. Numeric class: numeric
3. Alphanumeric class: alphanumeric, alphanumeric edited, numeric edited

Every elementary item is classified into one of these classes and categories by its PICTURE clause. Elementary items that do not have a PICTURE clause are in the numeric category.

Group items always belong to the alphanumeric category regardless of the categories of any elementary items they contain. A group item may be used in any place an alphanumeric item is allowed.

5.1.4 Standard Alignment Rules

The *standard alignment rules* define how characters are positioned in a data item when the item is receiving data. Positioning depends on the category of the *receiving* item.

1. For a numeric receiving item, the data is aligned by decimal point. Truncation or zero fill occurs as necessary. If no decimal point is explicitly stated, then the item is treated as if it had a decimal point after its rightmost character.

2. For a numeric edited item, the data is aligned by decimal point with zero fill or truncation as needed. Editing requirements can replace leading zeros with some other symbol.

3. For alphabetic, alphanumeric, and alphanumeric edited items, the data is aligned at the leftmost character position in the item. Space fill or truncation occurs on the right as needed.

The JUSTIFIED clause can change the standard alignment rules. For details, see the JUSTIFIED clause in **section 5.7.1.12**. The "--TruncANSI" compile option alters the truncation rules for COMP-5 items. See **Section 2.2.10.1** in Book 1.

5.1.5 Table Handling

Tables of data are common components of business data processing problems. You define tables of data items in COBOL by including the OCCURS clause in their data description entries. This clause specifies that the item is to be repeated as many times as stated. The item is considered to be a table element, and its name and description apply to each repetition or occurrence. Because each occurrence of a table element does not have a unique data name assigned to it, you can refer to a desired occurrence only by specifying the data-name of a table element, along with the occurrence number of the desired element. The occurrence number is known as a *subscript*.

The number of occurrences of a table element may be specified as fixed or variable. Although the number of occurrences of a table may be variable, the physical size of the table in computer memory is always fixed.

To define a one-dimensional table, use an OCCURS clause as part of the data description of the table element. The OCCURS clause must not appear in the description of group items that contain the table element. The following example shows a one-dimensional table defined by the item TABLE-ELEMENT.

```
01  TABLE-1.
    03  TABLE-ELEMENT OCCURS 20 TIMES.
        05  SUB-ELEMENT-1 ...
        05  SUB-ELEMENT-2 ...
```

In the preceding example, the complete set of occurrences of TABLE-ELEMENT has been assigned the name TABLE-1. However, you need not give a group name to a table unless you want to refer to the complete table as a unit. In the preceding example, TABLE-ELEMENT, SUB-ELEMENT-1, and SUB-ELEMENT-2 are all repeated data items and require subscripts.

Defining a one-dimensional table within each occurrence of an element of another one-dimensional table gives rise to a two-dimensional table. For example:

```
01  TABLE-2.
    03  BAKER OCCURS 20 TIMES.
        05  CHARLIE ...
        05  DOG OCCURS 5 TIMES ...
```

In the preceding example, DOG is a two-dimensional table; BAKER and CHARLIE are both one-dimensional.

Repeat this pattern to form multi-dimensional tables. Tables may have no more than 15 dimensions.

5.1.6 Large Data Handling

All COBOL data items may be larger than 64 KB in size, with some minor limitations:

- A file's maximum record size is 64 MB, so no data item in the FILE SECTION may exceed 64 MB.
- A data item with a size greater than 64 KB may not be given a VALUE phrase.

Data items larger than 64 KB may be used with most verbs, and data items larger than 64 KB may be reference modified.

Note: Earlier versions of ACUCOBOL-GT have different limitations. Refer to the appropriate version of your ACUCOBOL-GT documentation for details.

5.1.7 File Types

ACUCOBOL-GT manages four types of file organization. These are:

1. **Sequential Files** - are ordered by the historical order in which records are written to the file.
2. **Relative Files** - contain records that are identified by their record number, where the first record in the file is record number one. Relative files are ordered by ascending record numbers.
3. **Indexed Files** - contain records that have one or more *key fields*. Records in an indexed file are ordered by ascending values in these key fields. Each key of an indexed file represents an ordered sequence by which the records can be accessed. One of the key fields, known as the *primary key*, must contain a unique value for each record and is used to identify records uniquely.
4. **Sort Files** - are used only by the SORT, MERGE, RELEASE, and RETURN verbs. These are used to sort and merge records.

There are also four record types. These are:

1. **Fixed-length Records** - these records are a constant size.
2. **Variable-length Records** - these records contain information about the length of each record, which may vary.
3. **Text Records** - are sequential file records that contain text data. Text files should generally contain only USAGE DISPLAY fields, because the binary information contained in other types of fields may inadvertently resemble line-length delimiters used by the host computer system (e.g., carriage return or line-feed characters). Text records may optionally have trailing spaces automatically removed from them by the runtime system. This is determined by the device type named in the file's ASSIGN phrase.

4. **Print Records** - are text records that additionally contain printer carriage control information. Only sequential files may be print files. Print records have trailing blanks removed from them when they are written to the file. This is done to improve printing performance for printers that use serial communications. Unless otherwise noted, a print file follows the same rules as a text file.

The organization of a file is determined by the file's SELECT clause of the Environment Division and its FD or SD clause of the Data Division. The record type is determined by the first of the following rules that applies:

1. If the file's ASSIGN clause has the PRINT option, print records are used.
2. If any WRITE statement that references the file contains the ADVANCING phrase, print records are used.
3. If LINAGE is specified for the file, print records are used.
4. If PRINT-CONTROL is specified, print records are used.
5. If the file's SELECT has the LINE SEQUENTIAL clause, text records are used.
6. If the file's device type is DISPLAY or KEYBOARD, text records are used.
7. In RM/COBOL compatibility mode, if the file is sequential and "-Cb" is *not* specified, text records are used.
8. If the file's FD or SD contains a RECORD clause, variable-length records are used if the IS VARYING IN SIZE phrase is used or if both a minimum and maximum record size is specified. If only a single record size is specified, then fixed-length records are used.

Note: The compiler has an internal restriction of at least 6 bytes for SORT FILE records. If a record is shorter than that, the compiler detects it and pads the record to 6 bytes. Note also that in versions prior to 5.0, using SORT FILE with records shorter than 6 bytes would cause crashes.

---

9. If multiple record layouts are declared for the file and these records are not all the same size, variable-length records are used.

10. If the "-Cf" flag is used, then any variable-length record is made fixed-length.

11. If none of the preceding rules applies, then fixed-length records are used.

5.1.8 Floating-Point Data

A floating-point item is a numeric data item that allows for a very wide range of values. However, compared to other numeric data types in COBOL, floating-point data is less accurate. Most computer languages use floating-point to represent non-integer values. This makes floating-point a good method for sharing non-integer data with these other languages.

Floating-point data items differ in several ways from normal numeric data items:

- Floating-point items are stored in a machine-dependent format. In particular, they are stored in a format that is "native" to each machine. There are many floating-point formats currently in use by different machines, so floating-point data should not be considered portable.

- Floating-point items do not have pictures associated with them. Instead, floating-point items are either 4 or 8 bytes in size. The size of the item determines the range of values it can hold.

- The range of values that can be stored in a floating-point item is machine-dependent.

Because floating-point items do not maintain accuracy very well, you should limit their use. Some examples where floating-point is appropriate are:

- You need to share non-integer data with another language such as C or FORTRAN.

- You need to hold very large or very small values that exceed the usual 18 digits available in COBOL.

- You need to process existing data that contains floating-point values.

5.1.8.1 Using floating-point data

Generally speaking, you may use a floating-point data item anywhere that you can use a non-integer data item. Data moved to or from a floating-point item is converted to the appropriate format.

If you use a floating-point item in an arithmetic expression, then that expression is computed by converting all the values to double-precision floating-point and doing the arithmetic using the machine's conventions for double-precision math. The result is then converted to the type appropriate for the destination.

5.2 Data Names

The programmer assigns data names to COBOL data items in order to refer to those items in the program. Data names typically must be unique so that the compiler can know which data item the programmer is referring to. Data names that do not uniquely identify a data item may be made unique through *qualification* and *subscripting*. The programmer may also create new data names by *reference modification*. These three techniques are explained in the next three subsections.

5.2.1 Qualification

Every user-defined name explicitly referenced in a COBOL program must be uniquely defined in one of these ways:

1. No other name has the same spelling and hyphenation.
2. The name is unique within the context of a REDEFINES clause.
3. The name exists within a hierarchy of names, and reference to the name can be made unique by mentioning one or more of the higher level names in the hierarchy.

These higher-level names are called qualifiers. Identical user-defined names may appear in a source program; however, uniqueness must then be established through qualification for each user-defined name explicitly referenced. All available qualifiers need not be referenced as long as uniqueness is established.

General Format

Format 1

```
{data-name-1} { {OF} name-2 } ... [ {OF} file-name ]
{cond-name  }   {IN}               {IN}
```

Format 2

```
{data-name-1} {OF} file-name
{cond-name  } {IN}
```

Format 3

```
paragraph-name {OF} section-name
               {IN}
```

Format 4

```
lib-name {OF} dir-name
         {IN}
```

Format 5

```
LINAGE-COUNTER {OF} file-name
               {IN}
```

Syntax Rules

1. For each non-unique user-defined name that is explicitly referenced, uniqueness must be established through a sequence of qualifiers that precludes any ambiguity.
2. A name may be qualified even though it does not need qualification.
3. IN and OF are equivalent.

4. In Format 1, each qualifier must be the name associated with a group item to which the item being qualified is subordinate, or the name of a condition-variable with which the condition-name being qualified is associated. Qualifiers are specified in the order of successively more inclusive levels in the hierarchy.

5. If the program contains explicit references to a *paragraph-name*, the *paragraph-name* cannot appear more than once in the same section. A *paragraph-name* need not be qualified in a reference from within the same section that contains *paragraph-name*.

6. The LINAGE phrase of a file's FD creates an implicit data item called LINAGE-COUNTER. If more than one file in a program contains a LINAGE phrase, then reference to a file's LINAGE-COUNTER must be qualified by the name of the file.

7. If both qualification and subscripting are used in a data reference, the qualification is done first.

8. If both qualification and reference modification are used in a data reference, the qualification is done first.

9. A Format 4 form of qualification is used with the COPY statement. It is described in that section.

5.2.2 Subscripting

Subscripting is used when reference is made to an individual element of a table.

General Format

```
{data-name     }  ( { index-val [ {+} integer  ] } ...)
{condition-name}                  {-}
```

Syntax Rules

1. *Index-val* must be either an integer literal, an integer elementary data item, or an index name. It may be qualified.

2. *Data-name* and *condition-name* must be subordinate to an OCCURS clause.

3. *Integer* must be an integer literal.

4. The number of subscripts must equal the number of OCCURS clauses in the description of the table element being referenced. When more than one subscript is required, they are written in the order of successively less inclusive dimensions of the table.

5. If both qualification and subscripting of a data name are being used, the qualification is done first.

6. If both subscripting and reference modification of a data name are being used, the subscripting is done first.

General Rules

1. The value of the subscript must be a positive integer. The lowest occurrence value is represented by the value "1". Each successive element of a table within a dimension is referenced by occurrence numbers of 2, 3, 4, and so on. The highest permissible occurrence number for any given dimension of a table is the maximum number of occurrences of the item as specified by the associated OCCURS clause.

2. If *integer* is specified, the value of the subscript is determined by adding or subtracting the *integer* from *index-val*. This modified value is subject to all of the conditions of rule 1 above.

3. By default, it is not an error to reference a table element beyond the last one in the table, but the results are undefined and may adversely affect your program. In fact, this is the single most frequent cause of "memory access violation" errors. Use the "-Za" compiler option to cause an error message to appear whenever an out-of-bounds table element is referenced. The error text is: "Index out of bounds." This error is an *intermediate* runtime error that can trigger the execution of installed error procedures. See the entry for **CBL_ERROR_PROC** in Book 4, Appendix I.

5.2.3 Reference Modification

Reference modification is a syntax for referencing a portion (substring) of a data item. The reference defines a temporary, unique data item. Reference modification may be used anywhere in the Procedure Division.

Note: This manual entry includes code examples and highlights for first-time users following the General Rules section.

General Format

```
data-name ( leftmost-position : [length] )
```

Syntax Rules

1. *Leftmost-position* and *length* are arithmetic expressions.
2. Unless otherwise specified, reference modification is allowed anywhere a data item of the class alphanumeric is permitted.
3. *Data-name* may be qualified or subscripted. Reference modification is done after both qualification and subscripting.

General Rules

1. Each character of the data item referenced by *data-name* is assigned an ordinal number starting at one for the leftmost position and incrementing by one for each character in the item.
2. Reference modification for an operand is evaluated as follows:
   a. If subscripting is specified, the reference modification is evaluated immediately after the evaluation of the subscripts.
   b. If subscripting is not specified, the reference modification is evaluated at the time subscripting would have been evaluated if subscripts had been specified.
3. Reference modification creates a unique data item that is a subset of *data-name*. This unique data item is defined as follows:
   a. The evaluation of *leftmost-position* specifies the leftmost character of the unique data item relative to the start of *data-name*. Evaluation of *leftmost-position* must result in an integer greater than zero and less than or equal to the number of characters contained in *data-name*.

b. The evaluation of *length* specifies the size of the unique data item. The evaluation of *length* must result in a positive integer. The sum of *leftmost-position* and *length* must be less than or equal to the number of characters contained in *data-name*, plus one. If *length* is not specified, the unique data item extends through the rightmost character of *data-name*.

4. The unique data item is considered an elementary data item without the JUSTIFIED clause. It has the same class and category as *data-name* except that categories numeric, numeric edited, and alphanumeric edited are treated as class and category alphanumeric.

5. If the reference modification start or length parameter is out of range for the item it references, a runtime error occurs. How the runtime responds depends on the value of the WARNINGS configuration variable (see Book 4, Appendix I). By default, the runtime attempts to correct the error (see rule 6, below), and the warning message "Reference modifier range error" is displayed or sent to the error file. This error is an *intermediate* runtime error that can trigger the execution of installed error procedures (see the entry for CBL_ERROR_PROC in Book 4, Appendix I).

6. By default, the runtime silently corrects reference modification range errors by applying the following rules:

   a. A start reference less than 1 is treated as 1. For example, var(0:3) is treated as var(1:3).

   b. A length reference less than 0 is treated as 0. Moving a zero-byte item is equivalent to moving spaces to the destination item. A zero-byte destination is not affected by the move. In a STRING statement, a length of zero for a string source is treated as 1, not 0.

   c. A start plus length reference that is past the end of the item is treated as meaning to the end of the item. For example, if the var is a PIC X(5) item, var(4:23) is treated as var(4:2).

   The WARNINGS runtime configuration variable provides some control over how the runtime handles reference modification range errors. See the **WARNINGS** entry in Appendix H of Book 4.

Caution: Reference modification is allowed on *source-item* and *dest-item* of a Format 1 MOVE statement. However, when reference modification is used, *source-item* and *dest-item* should not reference the same item (or memory location). See general rule 7 of the **MOVE Statement** in Chapter 6.

Code Examples

Reference modification is akin to substringing in other programming languages. Reference modification is very useful for referencing a component part of a composite string. For example, it might be used to reference the area code digits of a 10-character string containing a phone number (area code + seven digits):

```
01  PHONE-NUMBER  PIC 9(10) VALUE 3017728134.
{ . . . }
PHONE-NUMBER (1:3).
*The reference modification begins at position 1
*of string PHONE-NUMBER and has a length of 3.
*The reference modification value = "301"
```

For the following code examples, assume these data items:

```
01 ACCOUNT-CODE PIC X(20) VALUE "AB700648xSMITHxxCLA1".
01 ACCOUNT-NAME PIC X(6)  VALUE ALL SPACES.
01 ACCT-CLASS-1 PIC X(4)  VALUE "CLA1".
```

Code example 1:

```
MOVE ACCOUNT-CODE (10:6) TO ACCOUNT-NAME.
*This reference modification selects the
*characters that form the name portion of
*ACCOUNT-CODE.  The reference starts at position
*10 and has a length of 6 characters.
*The ACCOUNT-CODE substring = "SMITHx"
```

Code example 2:

```
IF ACCOUNT-CODE (17:) = ACCT-CLASS-1 THEN
*When the reference modification does not
*include a length, the reference begins at the
*value specified and extends to the end of the
*data item.
*The ACCOUNT-CODE substring = "CLA1"
```

Highlights for first-time users

1. Reference modification may be used anywhere in the program where an alphanumeric data item may be referenced.
2. A reference modification does not create a persistent data item. Unless the result of the reference modification is assigned to a compatible data object, you can refer to the value of the reference modification later in the program only by repeating the reference modification.
3. The reference-modified data item is treated as an alphanumeric field.

5.2.4 Condition-Name (Level 88)

Level-number 88 designates a *condition-name* entry. Level 88s are used to assign names to values at execution time. Thus, a *condition-name* is not the name of an item, but rather the name of a value. A level 88 doesn't reserve any storage area.

Each level 88 must be associated with a data item and must immediately follow that item in the Data Division. The associated data item is called a *condition-variable*. A level 88 may name a specific value, a set of values, or a range of values. For example:

```
05  student-status    pic 9(2).
    88  kindergarten  value 0.
    88  elementary    values are 1 through 6.
    88  jr-high       values 7, 8, 9.
    88  high-school   values are 10 through 12.
```

Condition-names are often used in the Procedure Division as a test (usually with an IF statement) to specify conditions under which control will pass to another part of the program. They can make sentences much more meaningful to the reader. For example, if you've defined the condition-names shown above, then you could write this code:

```
if kindergarten
    perform assign-half-day-schedule.
```

Without the condition-name, you would have to write:

```
if student-status = "0"
    perform assign-half-day-schedule.
```

If you defined this condition-name:

```
07 priority-code        pic x.
   88 highest-priority  value "d".
```

then you could write this easily understood code:

```
if highest-priority perform fill-order-at-once.
```

Without the condition-name, you would have to write:

```
if priority-code = "d" perform fill-order-at-once.
```

Thus, real benefit comes from choosing a meaningful name for each value or set of values.

Setting a *condition-name* to TRUE is equivalent to moving any one of its values to the associated *condition-variable*. For example, note how the SET verb is used below to establish the truth of the condition:

```
05  end-of-shipping-file    pic x value "n".
    88  no-more-shipments   value "y".

...

perform process-daily-arrivals
  until no-more-shipments.

...

read shipping-file
  at end set no-more-shipments to true.
```

The same result could have been achieved with this code:

```
read shipping-file
  at end move "y" to end-of-shipping-file.
```

If explicitly referenced, a condition-name must be unique or must be made unique through qualification or subscripting. If qualification is used to make a condition-name unique, the associated condition-variable may be used as the first qualifier. The hierarchy of names associated with the condition-variable may be used in further qualification. If references to a condition-variable require subscripting, then references to the associated condition-name also require the same combination of subscripting.

For more information about *condition-names*, see **section 5.7.1, "Data Description Entry," section 5.7.1.14, "VALUE clause,"** and the **SET Statement** section.

5.2.5 RECORD-POSITION

The RECORD-POSITION construct allows you to refer to a data item by creating a numeric literal representing the location of the data item within a record.

General Format

```
RECORD-POSITION OF data-name
```

Syntax Rules

1. Data-name must refer to a data item with a level number of 01 through 50 or 77. Data-name may be qualified, but may not be subscripted or reference modified.
2. The RECORD-POSITION phrase is allowed anywhere a numeric literal data item may appear.

General Rules

1. The RECORD-POSITION phrase creates a numeric literal whose value is the character position of data-name within its record, as follows:
   a. If data-name is a level 01 or 77 data item, then the value is "1".
   b. Otherwise, the value is the character position of the start of data-name within its containing level 01 group item. Character positions start numbering at "1".
2. If data-name refers to a table item, the value is computed from the first occurrence of that item.
3. The format of the resulting literal is the same as a PIC 9(9) DISPLAY data item.

Code Examples

Code example 1:

If you assume the following group item:

```
01  GROUP-1.
    03  ELEM-1  PIC X(10).
    03  ELEM-2  PIC X(10).
    03  GROUP-2.
        05  ELEM-3
            OCCURS 10 TIMES  PIC X(10).
        05  ELEM-4  PIC X(10).
```

the following procedure division code:

```
DISPLAY RECORD-POSITION OF ELEM-1, CONVERT, LEFT
DISPLAY RECORD-POSITION OF ELEM-2, CONVERT, LEFT
DISPLAY RECORD-POSITION OF ELEM-3, CONVERT, LEFT
DISPLAY RECORD-POSITION OF ELEM-4, CONVERT, LEFT
```

would produce the following output:

```
1
11
21
121
```

Code example 2:

The RECORD-POSITION construct is particularly useful with the DATA-COLUMNS property of the list box and grid controls. For example, in a list box control, you might have a line that reads:

```
data-columns - (1, 13, 24, 33 )
```

changing the line to:

```
data-columns = (
    record-position of data-key-1,
    record-position of data-city,
    record-position of data-state,
    record-position of data-amount )
```

makes it easier to understand. With this syntax, changes to the record format do not need to be echoed in the data-columns format, so this is also easier to maintain.

5.3 Data Division Format

General Format

```
[ DATA DIVISION. ]

[ FILE SECTION.
   [ file-desc { record-description } ... ] ... ]
   [ sort-desc { record-description } ... ]

[ WORKING-STORAGE SECTION.
   [ record-description ] ... ]

[ LINKAGE SECTION.
   [ record-description ] ... ]

[ SCREEN SECTION.
   [ screen-description ] ... ]
```

Syntax Rules

1. The division header is optional for the Data Division.
2. The FILE SECTION header is optional.

General Rules

The Data Division entries are described in the following sections.

1. The File Section defines the structure of data files.
2. A *file-desc* entry and its associated *record-descriptions* specify the format, layout, and sizes of a file's logical records. A *sort-desc* entry specifies the layout and sizes of a sort file's logical records.
3. For each file described by a SELECT in the Environment Division, a corresponding *file-desc* or *sort-desc* must be made in the Data Division.

4. The Working-Storage Section describes the records and independent data items that are not part of data files but are developed and processed by the program internally.

5. Each *record-description* in Working Storage describes the format, layout, and size of an internal data item.

6. Data items in Working Storage can be given initial values (see **VALUE clause**). Items that are not explicitly initialized are set to spaces, or the value specified with the "-Dv" compile option, when the program is in its initial state. This may or may not be a valid value for the data item.

7. The Linkage Section is used only in a called program. It defines the data available from the calling program. Both the called and calling program can use this data.

8. To access data described in the Linkage Section, the called program may specify a USING phrase in its Procedure Division header. An alternative way to do this is through the SET ADDRESS OF statement. In the example below, note that the USING phrase has been omitted from the Procedure Division header.

```
LINKAGE SECTION.
01  my-var       pic x(30).

PROCEDURE DIVISION.
main-logic.
    if switch-1
       set address of my-var to msg-1
    else
       set address of my-var to msg-2
    end-if.
    display my-var.
```

   See **section 6.6** for additional information on the **SET Statement**.

9. The Screen Section describes the format, layout, and behavior of console screen items. These screen items are used with the ACCEPT and DISPLAY verbs to perform single- and multi-field console I/O.

5.4 File Section

The File Section describes the record-level information about the files that the program uses.

General Format

```
FILE SECTION.

   [ file-desc { record-description } ... ] ...
   [ sort-desc { record-description } ... ]
```

General Rules

1. The File Section header is followed by a series of *file-desc* entries and *sort-desc* entries.
2. A *file-desc* entry consists of a level indicator (FD), a file name, and a series of independent clauses. These clauses specify various logical and physical record attributes. They are described fully in the following sections.
3. A *sort-desc* entry consists of a level indicator (SD), a file name, and a series of independent clauses. These clauses specify various record attributes, which are described fully in the following sections.

5.4.1 File Description Entry

A file description entry describes the physical structure, identification, and record names for a program's data files.

General Format

```
FD file-name [IS EXTERNAL] [IS GLOBAL]

   [BLOCK CONTAINS [min TO] max  {RECORDS   } ]
                                 {CHARACTERS}

   [RECORD { CONTAINS [min TO] max CHARACTERS  } ]
           { IS VARYING IN SIZE [ FROM min ]   }
           {   [ TO max ] CHARACTERS           }
           {   [ DEPENDING ON depend ]         }
```

```
[ LABEL  { RECORD IS   }  {STANDARD} ]
         { RECORDS ARE }  {OMITTED }

[ VALUE OF LABEL IS label-lit ]

[ VALUE OF { FILE-ID } IS id-name ]
           { ID      }

[ CODE-SET IS alphabet ]

[ DATA  { RECORD IS   }  {record-name} ... ]
        { RECORDS ARE }

[ LINAGE IS page-size LINES

  [ WITH FOOTING AT footing-line ]

  [ LINES AT TOP top-lines ]

  [ LINES AT BOTTOM bottom-lines ] ] .
```

Syntax Rules

1. *File-name* must refer to a file name contained in a SELECT clause in the Environment Division.
2. The clause IS EXTERNAL must immediately follow the file-name in each program that declares the file.
3. An external file must have the same file-name in each COBOL program that declares it, and must be described the same way in each program.
4. The other clauses following *file-name* can appear in any order.
5. One or more record description entries must follow a file description entry.
6. The LINAGE phrase may be specified only for an FD associated with a sequential file.

General Rule

The file description entry clauses are described separately in the following sections.

5.4.2 Sort File Description Entry

A sort file description entry describes the physical structure for a sort file.

General Format

```
SD file-name
   [ RECORD { CONTAINS [min TO] max CHARACTERS } ]
            { IS VARYING IN SIZE [ FROM min ]  }
            {        [ TO max ] CHARACTERS     }
            {        [ DEPENDING ON depend ]   }

   [ DATA { RECORD IS   } {record-name} ... ]
          { RECORDS ARE }

   [ VALUE OF FILE-ID IS id-name ]
```

Syntax Rules

1. *File-name* must refer to a file name contained in a SELECT clause in the Environment Division. That SELECT clause may contain only ASSIGN and FILE STATUS clauses. However, a PASSWORD clause, TRACK-AREA clause, PROCESSING MODE clause, RECORDING MODE clause, FILE-LIMIT clause, VALUE OF clause, and APPLY clause may appear when the compiler is in the IBM DOS/VS COBOL compatibility mode. These phrases are scanned, but otherwise they are ignored.

2. The clauses following *file-name* may appear in any order.

3. *Id-name* must be either a non-numeric literal or the name of an alphanumeric data item in Working Storage. The value of this name is used as the file's external name.

4. If a *file-spec* is specified in the file's ASSIGN clause, *id-name* must be identical to *file-spec*.

5. One or more record description entries must follow a sort file description entry.

General Rules

1. No I/O statement may refer to a file described by a sort file description entry. Only the SORT and MERGE statements may refer to *file-name*.
2. The sort file description entry clauses are described in the following sections.

Note: The compiler has an internal restriction of at least 6 bytes for SORT FILE records. If a record is shorter than that, the compiler detects it and pads the record to 6 bytes.

5.4.3 IS EXTERNAL Clause

The IS EXTERNAL clause specifies that the file is shared by more than one program in a run unit.

General Format

```
IS EXTERNAL
```

Syntax Rules

1. The IS EXTERNAL declaration must be made in all the programs that access the file or item externally. The file must be declared external by each program in the run unit that will share a file's current state and record area.
2. The COBOL name of the file must be the same for all the programs that declare the file.
3. Each program that declares an external file must describe the file the same way.

General Rules

1. If one program opens the file, it is open for all programs that declare the same file.
2. If one program moves the record pointer, the record pointer moves in all the programs that declare the file.
3. Any data placed in the record area is accessible by all the programs that declare the file.

5.4.4 BLOCK CONTAINS Clause

The BLOCK CONTAINS clause specifies the size of a physical record.

General Format

```
BLOCK CONTAINS [min-block TO] max-block {RECORDS   }
                                        {CHARACTERS}
```

Syntax Rules

1. *Min-block* is an integer literal that specifies the minimum block size.
2. *Max-block* is an integer literal that specifies the maximum block size.

General Rules

1. The BLOCK CONTAINS clause specifies the physical record size.
2. The *min-block* specification is treated as commentary by the compiler. However, if RM/COBOL compatibility mode is being used, and *min-block* is specified, then the entire BLOCK CONTAINS clause is ignored by the compiler.
3. The compiler ignores the BLOCK CONTAINS clause for relative files.
4. For Vision files, *max-block* should be a multiple of 512 up to 8192 (the value is the block size in bytes). For Version 3 and 2 files, *max-block* should not exceed 1024. If it does, Vision automatically reduces it to 1024.
5. For sequential files, all input and output is done by blocks.

6. The RECORDS phrase specifies the physical record size in terms of logical records. If the file contains variable-length records, then the exact block size will vary from machine to machine depending on how variable-length records are stored on the host machine. The record size used to compute the block size is equal to the largest logical record.

7. The CHARACTERS phrase specifies the physical record size in terms of characters.

8. The final block of a file may contain fewer characters than specified by the BLOCK CONTAINS clause.

9. If no BLOCK CONTAINS clause is specified, the block size is set to one record. For files with variable-length records, the block size is set to the current record size (not necessarily the largest).

10. Records read from a file with variable-length records are internally blocked by ACUCOBOL-GT if no BLOCK CONTAINS clause is specified. This allows for efficient processing of these files on input while still allowing for line-by-line control over an output device (such as a printer).

Book 1, *User's Guide*, section 6.1, has more details on the handling of file blocking.

5.4.5 RECORD Clause

The RECORD clause describes the size of the logical records.

General Format

```
RECORD { CONTAINS [min-rec TO] max-rec CHARACTERS }
       { IS VARYING IN SIZE [ FROM min-rec ]      }
       {       [ TO max-rec ] CHARACTERS          }
       {       [ DEPENDING ON depend ]            }
```

Syntax Rules

1. *Min-rec* is an integer literal that defines the smallest record size.

2. *Max-rec* is an integer literal that defines the largest record size.

3. *Depend* is a numeric data item described in Working Storage or in Linkage. *Depend* cannot be in the File Section.

General Rules

1. This clause is never required, because the minimum and maximum record sizes of a file are computed by the compiler from the file's record descriptions. However, you may want to use this clause to indicate variable-length records.

   Note: The compiler has an internal restriction of at least 6 bytes for SORT FILE records. If a record is shorter than that, the compiler detects it and pads the record to 6 bytes.

2. No record description for a file can contain more characters than specified by *max-rec* or fewer characters than specified by *min-rec*.

3. If *min-rec* is omitted, it is set to be equal to *max-rec*.

4. If the VARYING phrase is used, then the file has variable-length records. If the CONTAINS phrase is used, and both *min-rec* and *max-rec* are specified, the file will contain variable-length records. If the CONTAINS phrase is used and only *max-rec* is specified, then the file will contain fixed-length records.

5. If the DEPENDING ON phrase is used, the size of the record written or rewritten to the file is set according to the value of *depend*. When a record is read from the file, *depend* is set to the size of the record found. Using the DEPENDING ON phrase automatically implies that the file has variable-length records.

Note: Other source statements may take precedence over the RECORD clause in determining the record type. The complete rules for determining a file's record type are described in **section 5.1.7, "File Types."**

5.4.6 LABEL RECORDS Clause

The LABEL RECORDS clause describes how file labels should be processed by the compiler.

General Format

```
LABEL  { RECORD IS   }  {STANDARD}
       { RECORDS ARE }  {OMITTED }
```

General Rule

This clause is ignored by the compiler.

5.4.7 VALUE OF LABEL Clause

This clause describes the values contained in a file's label records.

General Format

```
[ VALUE OF LABEL IS label-lit ]
```

Syntax Rule

Label-lit must be a non-numeric literal.

General Rule

This clause is ignored by the compiler.

5.4.8 VALUE OF FILE-ID Clause

The VALUE OF FILE-ID clause specifies the file's external name. (This can also be accomplished by the file's ASSIGN clause. The VALUE OF FILE-ID clause is provided for compatibility with other COBOL compilers.)

General Format

```
[ VALUE OF { FILE-ID } IS id-name ]
```

```
{ ID      }
```

Syntax Rules

1. *Id-name* is either a non-numeric literal or the name of an alphanumeric data item in Working-Storage. The value of this name is used as the file's external name.
2. If a *file-spec* is specified in the file's ASSIGN clause, *id-name* must be identical to *file-spec*.

5.4.9 CODE-SET Clause

The CODE-SET clause specifies the alphabet to use for a sequential file.

General Format

```
CODE-SET IS alphabet
```

Syntax Rules

1. *Alphabet* is the name of an alphabet declared in the SPECIAL-NAMES section.
2. The CODE-SET clause may be associated only with a sequential file.
3. If the CODE-SET clause is used, then only USAGE DISPLAY items may appear in the file's record description entry. Furthermore, every signed numeric field must have a SIGN IS SEPARATE clause.

General Rule

The CODE-SET clause associates a character set with a sequential file. If the character set is not the native character set, then a translation to the native set is also implied.

5.4.10 DATA RECORDS Clause

This clause names the record descriptions used by the file.

General Format

```
DATA { RECORD IS  } {record-name} ...
     { RECORDS ARE}
```

Syntax Rule

Record-name must name level 01 record descriptions associated with the file.

General Rule

This clause is never required, because the compiler can determine which records are associated with each file.

5.4.11 LINAGE Clause

The LINAGE clause is used to specify the number of lines on a logical page, and optionally provide margin information.

General Format

```
LINAGE IS page-lines LINES

   [ WITH FOOTING AT footing-line ]

   [ LINES AT TOP top-lines ]

   [ LINES AT BOTTOM bottom-lines ]
```

Syntax Rules

1. *Page-lines* is a numeric literal or numeric data item whose value must be a positive integer. It specifies the number of lines on the logical page. *Page-lines* may be qualified.

2. *Footing-line* is a positive integer or numeric data item. Its values must be greater than zero and less than or equal to *page-lines*. It specifies the line number where the footing area begins on the page. *Footing-line* may be qualified.

3. *Top-lines* and *bottom-lines* are integers or numeric data items. Their values must be greater than or equal to zero. They represent the numbers of lines in the top and bottom margins, respectively. *Top-lines* and *bottom-lines* may be qualified.

4. The LINAGE clause may be specified for a sequential file only.

General Rules

1. The LINAGE clause specifies the number of lines on a logical page. The total page size is the sum of *page-lines*, *top-lines*, and *bottom-lines*. If the TOP or BOTTOM phrase is omitted, the corresponding value is treated as zero. Note that *footing-line* is *not* added to the page size.

2. *Page-lines* specifies the size of the page body. This is the area of the logical page in which the program can write or space lines.

3. Each logical page follows the preceding logical page with no additional spacing. ACUCOBOL-GT does not provide physical page ejects (form-feeds) when the LINAGE clause is used. Device positioning occurs by line spacing rather than by page ejection.

4. The footing area is composed of the area between *footing-line* and *page-lines*, inclusive. The footing area causes a page-overflow condition when written in. If the FOOTING phrase is omitted, there is no footing area.

5. Evaluation of the logical page size occurs as follows:

   a. When the file is opened with the OUTPUT or EXTEND phrases, the LINAGE clause values are evaluated and applied to the first logical page. The device is assumed to be positioned at the beginning of the logical page.

   b. When the program executes a WRITE statement with the ADVANCING PAGE option, or when a page-overflow condition occurs, the LINAGE clause values are evaluated and applied to the next logical page.

6. For each file that has a LINAGE clause associated with it, the compiler creates an implicit data item called LINAGE-COUNTER associated with that file. If more than one file in the program specifies a LINAGE clause, reference to a LINAGE-COUNTER will have to be qualified by

the appropriate file name. The LINAGE-COUNTER is an elementary numeric data item that contains the same number of digits as *page-lines*.

7. At any time, the value of LINAGE-COUNTER is the line number in the current page body at which the device is positioned.

8. LINAGE-COUNTER may be treated as a normal numeric data item, except that it may never be explicitly modified by the program.

9. The LINAGE-COUNTER is set to one when the file is opened.

10. Every WRITE statement that refers to a file with a LINAGE clause affects the associated LINAGE-COUNTER in the following manner:

    a. If the WRITE statement has the ADVANCING PAGE phrase, the LINAGE-COUNTER is set to one.

    b. If the WRITE statement has the ADVANCING LINES phrase, the LINAGE-COUNTER is incremented by the value in the ADVANCING phrase.

    c. If the WRITE statement does not have an ADVANCING phrase, the LINAGE-COUNTER is incremented by one.

5.5 WORKING-STORAGE Section

The Working-Storage Section is used to define data items local to the program that do not reside in files.

General Format

```
WORKING-STORAGE SECTION.

   [ record-description ] ...
```

General Rules

1. Storage level 01, 77, and 78 data descriptions in Working-Storage must be unique since they cannot be qualified.

2. Subordinate data names need not be unique if they can be made unique through qualification.

3. Unless given a value by a VALUE clause, each data item defined in Working-Storage is initialized to spaces or the value specified with the "-Dv" compile option.

4. Level 01 and 77 data items in Working-Storage may be declared to be external, which means they are shared by more than one program.

   Each program of a run unit that declares an external data item may access that item. Any change to the item made by one program is automatically seen by all the other programs. External data items may be shared between COBOL and C programs.

5.6 LINKAGE Section

The Linkage Section is used to define data items that are passed from a calling program.

General Format

```
LINKAGE SECTION.

   [ record-description ] ...
```

General Rules

1. Each level 01 and 77 data item described in Linkage must be uniquely named.

2. Subordinate data names need not be unique if they can be made unique through qualification.

3. Each level 01 and 77 data item declared in Linkage should be named in a USING phrase of the Procedure Division header. Data items that are REDEFINES of other data items should *not* be named, however.

4. There is a limit of 255 level 01 Linkage data items per program. There is no limit for the number of subordinate items allowed for each of these level 01 items.

Note: There are two runtime configuration variables that relate to linkage items: **CHECK_USING** for specifying paramater size-matching testing, and **OPTIMIZE_INDIVIDUAL_LINKAGE** that perform address optimizations on each Linkage item individually.

5.7 Record Description Entry

A record description entry describes the name, size, and format of a COBOL logical record or data item.

General Format

```
{ data-description-entry } ...
```

Syntax Rule

The first data description entry of a record must have a level-number of 01 or 77 and start in Area A.

General Rules

1. Any data description entry that is not further subdivided is called an *elementary item*. A record itself may consist of an elementary item consisting of a single level 01 data description entry. A non-elementary entry is called a *group item*.
2. An elementary data item that is not part of a larger record must have a level-number of 01 or 77.
3. A group item that is not part of an enclosing group item must have a level-number of 01.

5.7.1 Data Description Entry

A data description entry specifies the attributes of one data item.

General Format

Format 1

```
level-number [ data-name ]
             [ FILLER    ]

   [ REDEFINES prev-data-name ]

   [ IS EXTERNAL      ]
   [ IS GLOBAL        ]
   [ IS SPECIAL-NAMES   {CURSOR        } ]
                        {CRT STATUS    }
                        {CHART STATUS  }
                        {SCREEN CONTROL}
                        {EVENT STATUS  }

[ IS EXTERNAL-FORM [ IDENTIFIED BY template-file-name ] ]
   [ IS IDENTIFIED BY external-name                     ]

   [ {PICTURE} IS picture-string ]
     {PIC    }

   [ [ USAGE IS ] usage-type ]

   [ [ SIGN IS ] {LEADING } [ SEPARATE CHARACTER ] ]
                 {TRAILING}

   [ OCCURS
         { table-size TIMES                         }
         { min TO max TIMES DEPENDING ON dep-item }

         [ {ASCENDING } KEY IS {key-name} ... ] ...
           {DESCENDING}

         [ INDEXED BY {index-name} ... ] ]

   [ {SYNCHRONIZED}  [LEFT ] ]
     {SYNC        }  [RIGHT]

   [ {JUSTIFIED} RIGHT ]
     {JUST     }

   [ BLANK WHEN ZERO ]
```

```
   [ VALUE IS value-lit ] .
```

Format 2

```
66 new-name RENAMES name-start [ {THRU   } name-end ] .
                                 {THROUGH}
```

Format 3

```
78 user-name VALUE IS {literal-1} [ {+} literal-2 ] .
                      {NEXT     }   {-}
                                    {*}
                                    {/}
```

Format 4

```
88 cond-name  {VALUE IS  } { low [ {THRU   } high ] } ...
              {VALUES ARE}         {THROUGH}

      [ WHEN SET TO FALSE false-val ] .
```

Syntax Rules

1. Level-number in Format 1 can be any number from 01 through 49, or 77.
2. Data description clauses may appear in any order, with two exceptions:
   a. The optional *data-name* or FILLER clause must immediately follow the level-number.
   b. The optional REDEFINES clause must immediately follow the *data-name* or FILLER clause.
3. Only level 01 and level 77 Working Storage data items may be declared EXTERNAL. Each program that declares an external data item must use the same name for that item, and the item must occupy the same number of bytes. External data items may not have a VALUE phrase.
4. The phrase IS GLOBAL is accepted by the compiler as a commentary.
5. *Template-file-name* and *external-name* are alphanumeric literals or unqualified data names. If a data name is used, it must refer to an unambiguous data item.

6. Format 1 data description entries that specify a PICTURE clause, or a USAGE clause that allows a PICTURE clause, are elementary items. All other Format 1 entries are group items.

7. There must be a PICTURE clause for all Format 1 elementary items except those that specify a USAGE clause that does not allow a PICTURE clause (for details, see **section 5.7.1.8, "USAGE clause"**).

8. The SYNCHRONIZED, PICTURE, JUSTIFIED, and BLANK WHEN ZERO clauses can appear only in a record description for an elementary item.

9. A level 78 entry associates a value with the name of a constant. *User-name* names the constant, and *literal* may be any literal.

10. Each *cond-name* requires a separate Format 4 entry. The level 88 entry associates a set of values with *cond-name*. All *cond-name* entries for a data item must immediately follow that data item's record description.

11. A *cond-name* can be associated with any data item except the following:

    a. another cond-name

    b. a level 66 or 78 data item

    c. a group item that contains items with JUSTIFIED, SYNCHRONIZED, or USAGE (other than USAGE DISPLAY) clauses.

    d. an index data item

General Rule

The individual clauses are described in the following sections.

5.7.1.1 Level-number

The level-number of a data item shows the hierarchy of that data item within its logical record. It is also used to denote individual data items and condition names.

Syntax Rules

1. Data description entries subordinate to an FD entry must have level-numbers with the values 01 through 49, 66, 78, or 88.
2. Data description entries in Working-Storage may have level-numbers with the values 01 through 49, 66, 77, 78, or 88.
3. Data description entries in Linkage may have level-numbers with the values 01 through 49, 66, 77, 78, or 88.

General Rules

1. The level-number 01 indicates the first entry in a record description.
2. Each data description entry in a record description can be subdivided into multiple data description entries, each having the same level-number. This level-number must be greater than the level-number of the subdivided entry, but less than 50. These level-numbers do not have to be successive. Thus you construct a record hierarchy of data items by using level-numbers.
3. An elementary data item may not be immediately followed by a data item with a higher level-number (except for levels 66, 77, 78, and 88).
4. Multiple level 01 entries subordinate to a file description entry represent implicit redefinitions of the same area.
5. Level-number 66 may be used to identify only RENAMES entries (Format 2 data description entry).
6. Level-number 77 identifies a non-contiguous data item entry. Level-number 77 entries may not have subordinate data description entries except for level 88 items.
7. A level 78 entry associates a value with the name of a constant. The name of the constant can be used anywhere the corresponding literal can be used.
8. You may use a level 78 named constant as a repeat count in a PICTURE string. This means that, in a PICTURE string, you may substitute a level 78 for a number in parentheses. In the following example, DATA-1 and DATA-2 are both the same size:

```
78  LENG-20     VALUE 20.
```

```
01  DATA-1      PIC X(20).
01  DATA-2      PIC X(LENG-20).
```

The PICTURE string that results from the substitution of the level 78 for its value must be legal. Use this in programs when several data items must be the same size and you want to be able to easily change the size in the future.

9. Level-number 88 defines a condition name. It can be used only in a Format 4 data description entry.

5.7.1.2 The data-name or FILLER clause

The data-name clause specifies the name of the data area.

General Format

```
[ data-name ]
[ FILLER    ]
```

Syntax Rules

1. The *data-name* or FILLER clause must appear immediately after the level-number in a data description entry.
2. *Data-name* is a user-defined word.
3. A FILLER data item may not be declared EXTERNAL.

General Rules

1. If there is no *data-name* or FILLER clause, the FILLER clause is implied.
2. The FILLER clause names a data item that the program cannot explicitly refer to.
3. The FILLER clause can name a data item that is the object of a level 88 condition name.

5.7.1.3 REDEFINES clause

The REDEFINES clause allows the same computer memory area to be described by different data items. ACUCOBOL-GT extends ANSI-85 COBOL by allowing a REDEFINES phrase to reference an item that is itself a redefinition of an area.

General Format

```
level-number  [ data-name ]  REDEFINES  prev-data-name
              [ FILLER    ]
```

Syntax Rules

1. The *level-number*, *data-name*, and FILLER phrases in the General Format are not actually part of the REDEFINE clause. They are included for clarity.

2. The level-numbers of the subject of a REDEFINES clause and *prev-data-name* must be the same. They may not be 66 or 88.

3. REDEFINES is allowed in a level 01 entry in the File Section, but it will generate a warning message.

4. The number of character positions described by *prev-data-name* need not be the same as the number of character positions in the subject of the REDEFINES clause. The compiler generates a warning, however, if the number of character positions is greater in the subject of the REDEFINES clause than in *prev-data-name* and the level-number of the two data items is not 01 or 77 (this case is not allowed under ANSI COBOL).

5. The data item being redefined may be qualified, but any qualification specified is ignored.

   Example: 01 MY-FILLER REDEFINES THIS-FIELD OF THIS-GROUP.

   The phrase in the example compiles, but the qualification “OF THIS-GROUP” is ignored.

6. Several data items can redefine the same memory area.

7. No entry with a level-number lower than that of *prev-data-name* can occur between the data description entry for *prev-data-name* and the redefinition.

8. All entries redefining the storage area of a data item must immediately follow the entries describing that data item. No intervening entries that define additional storage may appear.

9. The IS EXTERNAL clause may not be used with the FILLER or REDEFINES clauses.

General Rules

1. Storage allocation for the redefining data item starts at the location of *prev-data-name*.

2. Storage allocation continues until it defines the number of character positions described by the redefining entry.

3. *Prev-data-name* may contain the OCCURS clause, although this is not compatible with ANSI COBOL. If such a situation exists, the compiler will return a "caution" warning indicating a non-ANSI construct. Cautions are shown only when you compile with the "-a" option. When you REDEFINE a data item with an OCCURS clause, the redefining item starts at the same memory location as the first occurrence of the redefined item.

4. In large model programs, certain REDEFINES could cause VALUE clauses to be lost. This happens when the VALUEs are set in a data item that is not a large data item, and then that data item is redefined as a large data item. When that occurs, the compiler detects the situation and issues a warning message:

   ```
   Warning: Large redefines of a regular variable with a
   value: desc2 redefines desc1
   ```

   When you see this warning message, you should modify your COBOL program to add FILLER to the first data item in order to make it a large data item. For example, the following code:

   ```
   01  small-group-item.
     03  small-data-item pic x(100) value "this is a test".

   01  large-group-item redefines small-group-item.
     03  free-form-text  pic x(100) occurs 1000 times.
   ```

will compile, but the value of small-data-item will be spaces when the program starts. To work around this, add:

```
03  filler   pic x(65000).
```

to the small-group-item after the small-data-item. The resulting code should look like this:

```
01  small-group-item.
  03  small-data-item pic x(100) value "this is a test".
  03  filler          pic x(65000).
```

5.7.1.4 IS EXTERNAL clause

The IS EXTERNAL clause declares that a data item is shared by two or more programs.

General Format

```
[ IS EXTERNAL ]
```

Syntax Rules

1. Only level 01 and 77 data items in Working-Storage may be declared to be external.
2. Each program that declares an external data item must use the same name for that item, and the item must occupy the same number of bytes.
3. External data items may not have a VALUE phrase.
4. The IS EXTERNAL clause may not be used with the REDEFINES clause.

General Rules

1. The phrase IS EXTERNAL must be included in the data description of the item in each program that accesses the item externally. Each program of a run unit that declares an external data item may access that item. Any change to the item made by one program is automatically seen by all the other programs. External data items may be shared between both COBOL and C programs.

2. An external data item belongs to the run unit, not to any of the programs that are part of the run unit. This means that an external data item is allocated for the duration of the run, regardless of the action of CANCEL verbs.

   The one exception to this rule is external data items that are also C data items. These belong to the runtime system itself, not to the run unit. The distinction is that you can initiate another run unit with the CALL RUN verb, but this does not initiate another runtime system. In this case, you get new copies of the COBOL-only external data items, but keep the same data items that are shared with C programs.

3. Instructions for declaring a C data item external can be found in the file "direct.c" supplied with the runtime system. External data items may not have a VALUE phrase. They are initialized to binary zeros (NULL) by the runtime.

5.7.1.5 IS SPECIAL-NAMES clause

The IS SPECIAL-NAMES clause allows you to identify select Special-Names directly in the Data Division.

General Format

```
[ IS SPECIAL-NAMES   {CURSOR         } ]
                     {CRT STATUS     }
                     {CHART STATUS   }
                     {SCREEN CONTROL }
                     {EVENT STATUS   }
```

Syntax Rule

The syntax is identical to declaring *data-name* in Special-Names with the indicated phrase. See **section 4.2.3**.

General Rule

Only one data item can be declared for each Special-Names type. Items identified in the IS SPECIAL-NAMES phrase cannot appear in the Special-Names paragraph of the Environment Division. If there is a corresponding item, the declaration must be identical.

The advantage of the IS SPECIAL-NAMES syntax over naming the items in Special-Names is that a single COPY library can be used to include all of your commonly used Special-Names items. For example, you might have a COPY library that reads:

```
* Declare commonly used screen handling items

01  SCREEN-CONTROL IS SPECIAL-NAMES SCREEN CONTROL.
    03  ACCEPT-CONTROL     PIC 9.
    03  CONTROL-VALUE      PIC 999.

01  CURSOR-POSITION IS SPECIAL-NAMES CURSOR.
    03  CURSOR-ROW         PIC 999.
    03  CURSOR-COL         PIC 999.

77  CRT-STATUS IS SPECIAL-NAMES CRT STATUS PIC 9(5).
```

5.7.1.6 IS EXTERNAL-FORM clause

The IS EXTERNAL-FORM clause associates a group item with HyperText Markup Language (HTML) data using the Common Gateway Interface (CGI) specification. It allows you to define input and output records for HTML forms and is useful when your COBOL code is part of an Internet-based application.

General Format

```
[ IS EXTERNAL-FORM [ IDENTIFIED BY template-file-name ] ]
[ IS IDENTIFIED BY external-name ]
```

Syntax Rule

Template-file-name and *external-name* are alphanumeric literals or unqualified data names. If a data name is used, it must refer to an unambiguous data item.

General Rules

1. The EXTERNAL-FORM clause associates a group item with HTML data using the Common Gateway Interface (CGI) specification. It allows you to define input and output records for HTML forms. The

EXTERNAL-FORM clause affects the way ACCEPT and DISPLAY process the data item. It does not put any restrictions on the way that the data item may be used in your program.

2. An EXTERNAL-FORM data item is called an "output form" if the IDENTIFIED BY clause is used in the description of the group item. This clause associates the data item with an HTML template file. If the IDENTIFIED BY clause is omitted from the group item, the EXTERNAL-FORM data item is called an "input form".

3. For "input forms," the IDENTIFIED BY clause establishes an association between an elementary data item and a CGI variable. The value of *external-name* is the name of the CGI variable. If the IDENTIFIED BY phrase is omitted, then data item's own name (*data-name*) is used as the name of the CGI variable. If both the IDENTIFIED BY phrase and *data-name* are omitted, then the data item has no corresponding CGI variable.

4. CGI variables are case-sensitive. The runtime matches CGI names according to their case. However, if a CGI variable is not found using a case-sensitive match, then the runtime tries a case-insensitive match. Note that *data-name* is always treated as if it were upper case regardless of the case used in the COBOL source. The case of the value specified by the IDENTIFIED BY phrase is preserved.

5. The ACCEPT verb treats input forms and output forms in the same manner. ACCEPT sets the value of each elementary item, in order, to the value of its associated CGI variable. The CGI data is retrieved from the program's standard input. ACCEPT automatically decodes and translates the CGI input data before moving it to the elementary data item. The value of each CGI variable is converted to the appropriate COBOL data type when it is moved. If the CGI variable is empty or does not exist, ACCEPT sets the value of numeric data items to zero, and nonnumeric data items to spaces.

6. To receive a CGI variable that is repeated (this occurs when multiple items have been selected in a "choose many" list), you should use an elementary data item that is part of a table. Each occurrence of the data item receives one of the repeated values. The first occurrence receives the first repeated CGI item; the second occurrence receives the second repeated item; and so forth. Occurrences that do not correspond to repeated CGI items are set to zero if the data item is numeric, or spaces otherwise.

7. Data items are matched to CGI variables immediately before the particular CGI data item is retrieved. Thus it is possible for a form to have CGI variable names supplied by the CGI input itself. Consider:

```
01  MY-FORM IS EXTERNAL-FORM
    03  CGI-VAR1 PIC X(20) IDENTIFIED BY "Name".
    03  CGI-VAR2 PIC X(50) IDENTIFIED BY CGI-VAR1.
```

   In this example, an ACCEPT MY-FORM statement would first locate the CGI variable called “Name” and move its value to CGI-VAR1. It would then locate a CGI variable identified by that value and move the corresponding value to CGI-VAR2. Note that, for this to work, you must specify CGI-VAR1 before CGI-VAR2 in MY-FORM, because ACCEPT updates the elementary data items in order.

8. The DISPLAY verb treats input and output forms differently. For output forms, DISPLAY merges the data contained in the elementary items into the associated HTML template file and sends the result to the standard output stream in conformance with the CGI specification. To do this, DISPLAY scans the HTML template file for data names delineated by two percentage signs on either side (i.e. %%data-name%%). It then replaces those data names with the contents of the associated elementary items from the output form, stripping trailing spaces. The maximum length of a single line in the template file is 256 bytes. There is virtually no limit to the length of a single HTML output line. No conversion is performed on the output form items before they are merged with the HTML template file.

9. When an input form is specified in a DISPLAY statement, the names and values of each elementary item are sent to the standard output stream in HTML format. One line is generated for each elementary item. The line consists of the name of the item followed by “ = “, followed by the first 100 bytes of the item’s value. This can be useful when you are testing and debugging your CGI program.

10. *Template-file-name* specifies the name of the HTML template file. You can specify a series of directories for locating HTML template files. To do this, use the HTML_TEMPLATE_PREFIX configuration variable. This variable is similar to FILE_PREFIX and CODE_PREFIX. It specifies a series of one or more directories to be searched for the desired HTML template file. The directories are specified as a sequence of space-delimited prefixes to be applied to the

file name. All directories in the sequence must be valid names. The current directory can be indicated by a period (regardless of the host operating system).

11. You can omit the suffix if it is either ".html" or ".htm". If the suffix is omitted or is something other than ".html" or ".htm", DISPLAY first appends ".html" to the specified file name and tries to open it. If that fails, DISPLAY appends ".htm" to the file name and tries to open it. If that fails, DISPLAY tries to open the file exactly as specified. If these attempts fail, the following error message is sent to the standard output stream in HTML format:

    ```
    Can't open HTML template "template-file-name"
    ```

12. When the Web Server executes your CGI program, the current working directory depends on the configuration of the specific Web Server that is running. In many cases it is the same as the Web Server's "root" directory. As part of the CGI specification, when the Web Server executes your CGI program, it sets an environment variable called PATH_TRANSLATED to the directory containing your CGI program. You may want to use this information to locate your HTML template files.

    For example, if your template files are in the same directory as your CGI programs, then set the HTML_TEMPLATE_PREFIX configuration variable to the value of PATH_TRANSLATED as follows:

    ```
    01  CGI-DIRECTORY     PIC X(100).

    ACCEPT CGI-DIRECTORY FROM ENVIRONMENT "PATH_TRANSLATED"
    SET CONFIGURATION "HTML_TEMPLATE_PREFIX" TO
       CGI-DIRECTORY.
    ```

13. The output from a CGI program must begin with a "response header". DISPLAY automatically generates a "Content-Type" response header when *template-file-name* specifies a local file (*i.e.*, not a URL - see rule #15 below).

14. You may specify the EXTERNAL-FORM clause for an item that has no subordinate items. This is useful for displaying static Web pages. To do this, specify the name of the static Web page in *template-file-name*. For example, if you have a Web page called "webpage1.html", add the following lines to your COBOL program:

```
01  WEB-PAGE-1 IS EXTERNAL-FORM,
          IDENTIFIED BY "webpage1".

DISPLAY WEB-PAGE-1.
```

15. You may also specify a complete URL in *template-file-name*. In this case, DISPLAY generates a "Location" response header that contains the URL. This header specifies that the data you're returning is a pointer to another location. To determine whether *template-file-name* is a URL, DISPLAY scans it for the string "://". DISPLAY does not apply HTML_TEMPLATE_PREFIX when *template-file-name* is a URL. For example, if your program determines that the information the user has requested is on another Web server, and its URL is "http://www.theinfo.com", add the following lines to your COBOL program:

    ```
    01  THE-INFO-URL IS EXTERNAL-FORM
          IDENTIFIED BY "http://www.theinfo.com"

    DISPLAY THE-INFO-URL.
    ```

    The length of the URL must not exceed 256 bytes. Only one response header is sent to the standard output stream. Your CGI program should exit immediately after sending a location header (*i.e.*, after displaying an external form identified by a URL).

16. You may use as many HTML template files as you like in a single program. A common way to use multiple HTML template files is to have three output forms: a header, body, and footer. Each of these has a corresponding HTML template file. You first display the header form, then move each row of data to the body form and display it, and finally display the footer form.

17. Data items that do not have EXTERNAL-FORM specified are treated as regular data items by ACCEPT and DISPLAY, even if they are subordinate to an external form. For example:

    ```
    01  MY-FORM IS EXTERNAL-FORM.
        03  CGI-VAR1  PIC X(10)
        03  CGI-VAR2  PIC 9(5).
    ```

    Using this data structure, an ACCEPT of MY-FORM would fill in CGI-VAR1 and CGI-VAR2 with CGI data. An ACCEPT of CGI-VAR1 would simply get data from the user just as it does for any regular data item.

5.7.1.7 PICTURE clause

The PICTURE clause describes the general characteristics and editing formats of an elementary item.

General Format

```
{PICTURE} IS picture-string
{PIC    }
```

Syntax Rules

1. A PICTURE clause can appear for elementary items only.
2. The maximum size of the *picture-string* is 100 characters.
3. A PICTURE clause is required for every elementary data item except for those items with a USAGE clause that disallows a PICTURE, or those items that are the subject of a RENAMES clause. A PICTURE clause is prohibited for these items.
4. PIC and PICTURE may be used interchangeably.
5. A picture is invalid if it specifies more than 31 digits to the left of the decimal point or more than 32 digits to the right of the decimal point, including assumed zero digits represented by "P" and floating insertion positions that may hold digits. If a picture exceeds these scaling limitations, a compile-time error message is produced.

General Rules

The PICTURE clause defines a data item as belonging to one of five categories and determines what the item can contain. The five categories are:

- Alphabetic
- Numeric
- Alphanumeric
- Alphanumeric Edited
- Numeric Edited

Alphabetic:

1. An item is *alphabetic* when its *picture-string* consists solely of "A" symbols.
2. An alphabetic item may contain only one or more alphabetic characters.

Note: The alphabetic declaration show's the programmer's intent to store only alphabetic data, however the system does not provide any checks to ensure compliance.

Numeric:

1. An item is *numeric* when its *picture-string* contains only the symbols "9", "P", "S", and "V". The number of digit positions described by *picture-string* must range from 1 to 18 inclusive. This increases to 31 digit positions if 31-digit support (-Dd31 compiler option) is in effect.
2. If unsigned, its contents must be one or more numeric characters. If signed, then the item may also contain a "+" or "-" (or other representation of the sign--see **section 5.7.1.9, "SIGN clause"**).
3. The numeric category of the PICTURE clause includes an external floating-point data item, which is defined by a picture that strongly resembles a floating-point numeric literal. For more information about External Floating-Point, see section 5.5 in *Transitioning to ACUCOBOL-GT.*

Alphanumeric:

1. An item is *alphanumeric* when its *picture-string* consists solely of the symbols "A", "X", and "9". A *picture-string* containing all "A" or all "9" symbols is *not* alphanumeric. When used in an alphanumeric item, the "A" and "9" symbols are treated as if they were "X" symbols.
2. Its contents may contain one or more characters in the computer's character set.

Alphanumeric edited:

1. An item is *alphanumeric edited* when its *picture-string* contains certain combinations of the symbols “A”, “X”, “9”, “B”, “0”, and “/”. The *picture-string* must contain at least one “A” or “X” symbol and at least one “B”, “0”, or “/” symbol.

2. Its contents may contain two or more characters in the computer’s character set.

Numeric edited:

1. An item is *numeric edited* when its *picture-string* contains certain combinations of the symbols “B”, “/”, “P”, “V”, “Z”, “0”, “9”, comma, period, “*”, “+”, “-”, “CR”, “DB”, and the currency symbol. The number of digit positions that can be represented by the *picture-string* must range between 1 and 18 inclusive. This increases to 31 digit positions if 31-digit support (-Dd31) is in effect. The *picture-string* must contain at least one “0”, “B”, “/”, “Z”, “*”, “+”, comma, period, “-”, “CR”, “DB”, or currency symbol.

2. The content of each character position must be consistent with the corresponding PICTURE symbol.

All types:

1. Some PICTURE symbols represent character positions and some do not. A data item’s size is determined by adding up all the symbols that represent character positions.

2. A *picture-string* may contain repeat counts for its symbols. You denote this by placing the repeat count in parentheses immediately after the symbol that is being repeated. For example, “X(4)” and “XXXX” are equivalent PICTURE strings.

3. Only one “S” may appear in a PICTURE string, and it must be the leftmost character. Only one of the following symbols may appear in a PICTURE string: “V” or the period character. It may appear only once. At least one of the symbols “A”, “X”, “Z”, “9”, or “*” or at least two occurrences of the symbols “+”, “-”, or the currency symbol must be present in a PICTURE string.

The PICTURE symbols and their functions are the following:

A Represents a character position that can contain only an alphabetic character. It is counted in the size of an item. An alphabetic character is any character “A” through “Z”, “a” through “z”, or a space.

B Represents a character position where a space will be inserted. It is counted in the size of an item.

E A delimiter directly preceding the (signed or unsigned) exponent part of an External Floating-Point notation. The exponent is always exactly two digits long, so the last two characters of the picture must be “99”. This delimiter is counted in the size of an item.

P This symbol is used to specify an assumed scaling position in a number. The “P” character is not counted in the size of the data item. Instead, each “P” represents a scaling of the data item by a power of ten. “P” elements can appear only as a contiguous string of “P”s in either the leftmost or rightmost digit positions of a *picture-string*. If the “V” character is used, it must be to the left of all leading “P”s or to the right of all trailing “P”s. The assumed decimal point for the item is located at the point where the “V” character may be placed. For example, the PICTURE string “9P” is a one-digit item that can have numeric values of “10”, “20”, ..., “90”, and a PICTURE string of “PPP9” is a one-digit item that can have numeric values of “0.0001” though “0.0009”. The “.” character may not be specified in the same *picture-string* as a “P” character. When a data item is treated as a numeric value, each “P” position acts as if it were a digit position that contained a zero. When a data item is not treated as a numeric value, then its “P” positions are ignored.

S Indicates the presence of an operational sign for a numeric item. It does not specify the sign representation or position. Only one "S" may appear in a PICTURE string, and it must be the leftmost character. It does not count in the size of the data item unless the SIGN IS SEPARATE clause is also specified.

Note that the PICTURE definition for a PIC X(n) COMP-5 cannot be signed.

V Specifies the location of an assumed decimal point. It may appear only once in a PICTURE string. It does not represent a character position and is not counted in the size of the data item.

X Represents a character position that can contain any character from the computer's character set. It is counted in the size of an item.

Z Represents a leading digit position that is replaced by a space when its value and the digit positions to its left are all zero. It is counted in the size of an item.

9 Represents a digit position which is counted in the size of an item.

0 Represents a digit position where a zero will be inserted. It is counted in the size of an item.

/ Represents a character position where a slash will be inserted. It is counted in the size of an item.

, The comma character represents a character position where a comma will be inserted. It counts in the size of an item.

. The period character represents a character position where a decimal point will be inserted. It also implies an operational decimal point for alignment purposes. It counts in the size of an item. Note that the functions of comma and period are exchanged if the DECIMAL-POINT IS COMMA clause is stated in the program's SPECIAL-NAMES paragraph.

+ - Represent editing sign control symbols. These can occur more than once. Each counts in the size of an item.

CR Represents an editing sign control symbol. It may be used only once on the rightmost side of the PICTURE string. It adds two to the size of a data item.

DB Represents an editing sign control symbol. It may be used only once on the rightmost side of the PICTURE string. It adds two to the size of a data item.

* Represents a leading digit that is replaced by asterisks when its value and all of the digits to its left are zero. It is counted in the size of an item.

$ Represents a character position into which the currency symbol is inserted. It is counted toward the size of an item. Note that the dollar sign is the default currency symbol. It may be changed by the CURRENCY clause of the SPECIAL-NAMES paragraph.

Editing Rules

1. The two methods of editing

   There are two general methods of performing editing in the PICTURE clause, either by *insertion* or by *suppression and replacement*.

2. The four types of insertion editing:

   a. Simple insertion

   b. Special insertion

   c. Fixed insertion

   d. Floating insertion

3. The two types of suppression and replacement editing:

   a. Zero suppression and replacement with spaces

   b. Zero suppression and replacement with asterisks

4. Types of editing allowed:

The type of editing that may be performed on an item is dependent on the category of that item. No editing may be performed on *alphabetic*, *numeric*, or *alphanumeric* items. Simple insertion of types "0", "B", and "/" may be performed on *alphanumeric edited* types. All forms of editing are allowed on *numeric edited* items.

5. Floating insertion, zero suppression, and the PICTURE clause

   Floating insertion and zero suppression and replacement are mutually exclusive in a PICTURE clause. Only one type of zero suppression and replacement may be used in a PICTURE clause.

6. Simple Insertion Editing

   The comma, "B", "0", and "/" are used as simple insertion characters. The insertion characters represent positions in an item where those characters will be inserted. If the comma is the last symbol in a PICTURE clause, the PICTURE clause must be the last clause in its data description entry, and the clause must be immediately followed by a period.

7. Special Insertion Editing

   The period is used as an insertion character, as it is in simple insertion. In addition to being used as an insertion character, it also represents the operational decimal point of the item. The period may not appear in the same PICTURE clause as the "V" symbol. If the period is the last symbol in the PICTURE clause, then the PICTURE clause must be the last clause in its data description entry, and the clause must be immediately followed by a period.

8. Fixed Insertion Editing

   The currency symbol and the editing sign control symbols "+", "-", "CR", and "DB" are the insertion characters. Only one currency symbol and only one of the editing sign control symbols can be used in a given PICTURE clause. The symbols "CR" and "DB", when used, must appear as the rightmost symbols in the PICTURE string. The symbol "+" or "-", when used, must either be the leftmost or rightmost character position to be counted in the size of the item. The currency symbol must be the leftmost character except that it may be preceded by either a "+" or "-" symbol.

Fixed insertion editing results in the insertion character's occupying the same character position in the edited item as it occupied in the PICTURE string. Editing sign control symbols produce the following results depending on the value of the data item:

Editing Symbol	Zero or Positive	Negative
+	+	-
-	space	-
CR	2 spaces	CR
DB	2 spaces	DB

9. Floating Insertion Editing

The currency symbol and editing sign control symbols "+" and "-" are the floating insertion characters. They are mutually exclusive in a PICTURE clause. You indicate floating insertion by using a string of at least two of the floating insertion characters. This string may contain any of the simple insertion characters or have the simple insertion characters immediately to the right. These simple insertion characters are part of the floating string.

The leftmost character of the floating insertion string represents the leftmost limit of the floating symbols in the data item. The rightmost character represents the rightmost limit. The second floating character from the left represents the leftmost limit of the numeric data that can be stored in the data item. Non-zero numeric data may replace all the characters at or to the right of this limit.

In a PICTURE string, there are only two ways of representing floating insertion editing. One way is to represent any or all of the leading numeric character positions on the left of the decimal point by the insertion character. The other way is to represent all of the numeric character positions by the insertion character.

If the insertion character positions are only to the left of the decimal point, the result is that a single floating insertion character will be placed into the character position immediately preceding either the decimal

point or the first non-zero digit in the data represented by the insertion symbol string, whichever is leftmost. The character positions preceding the insertion character are replaced by spaces.

If all numeric character positions in the PICTURE string are represented by the insertion character, the results depend on the value of the data. If the value is zero, the entire data item will contain spaces. If the value is not zero, then the results are the same as when the insertion character is only to the left of the decimal point.

The character inserted for the "+" symbol is "+" if the value is zero or positive and "-" if it is negative. The character inserted for the "-" symbol is "-" if the value is negative and space otherwise.

10. Zero Suppression and Replacement Editing

    The suppression of leading zeros in numeric character positions is indicated by the use of the "Z" and "*" symbols. These symbols are mutually exclusive in a PICTURE clause. If a "Z" is used, the replacement character is a space. If "*" is used, the replacement character is an asterisk.

    You indicate zero suppression and replacement by using a string of one or more of the allowable symbols to represent leading numeric character positions which are to be replaced when the associated character position in the data contains a leading zero. Any of the simple insertion characters embedded in the string of symbols or to the immediate right of this string are part of the string.

    In a PICTURE string, there are only two ways of representing zero suppression. One way is to represent any or all of the leading numeric character positions to the left of the decimal point by suppression symbols. The other is to represent all of the numeric character positions by suppression symbols.

    If the suppression symbols appear only to the left of the decimal point, any leading zero in the data that corresponds to a symbol in the suppression string is replaced by the replacement character.

    If all numeric positions in the PICTURE string are represented by suppression symbols and the value of the data is not zero, the result is the same as if the suppression characters were only to the left of the decimal

point. If the value is zero and the suppression symbol is “Z”, the entire data item is set to spaces. If the suppression symbol is “*” instead, the entire data item is set to asterisks except for the decimal point (if any), which will appear in the data item.

11. The symbols “+”, “-”, “*”, “Z”, and the currency symbol, when used as floating replacement characters, are mutually exclusive in a given PICTURE clause.

5.7.1.8 USAGE clause

The USAGE clause specifies the format of a data item in computer memory or in a file record.

In some circumstances, a data item’s file-record format may differ from its computer-memory format as specified by the USAGE clause. This can occur when non-COBOL file systems with different data storage formats are being accessed through an interface. For example, Acu4GL uses SQL to access non-COBOL file systems, and in the process a translation occurs on the data.

Note: There are numerous compiler options for affecting data storage behavior. See **Section 2.2.10, “Data Storage Options”** of the *ACUCOBOL-GT User’s Guide* for details on these options.

General Format

```
[ USAGE IS ] {COMPUTATIONAL                }
             {COMP                         }
             {COMPUTATIONAL-1              }
             {COMP-1                       }
             {COMPUTATIONAL-2              }
             {COMP-2                       }
             {COMPUTATIONAL-3              }
             {COMP-3                       }
             {COMPUTATIONAL-4              }
             {COMP-4                       }
             {COMPUTATIONAL-5              }
             {COMP-5                       }
             {COMPUTATIONAL-6              }
             {COMP-6                       }
             {COMPUTATIONAL-X              }
             {COMP-X                       }
```

```
{COMPUTATIONAL-N                                  }
{COMP-N                                           }
{BINARY                                           }
{PACKED-DECIMAL                                   }
{DISPLAY                                          }
{INDEX                                            }
{POINTER                                          }
{FLOAT                                            }
{DOUBLE                                           }
{SIGNED-SHORT                                     }
{UNSIGNED-SHORT                                   }
{SIGNED-INT                                       }
{UNSIGNED-INT                                     }
{SIGNED-LONG                                      }
{UNSIGNED-LONG                                    }
{HANDLE [ OF {WINDOW                        } ] }
             {SUBWINDOW                     }
             {FONT [font-name]              }
             {control-type                  }
             {THREAD                        }
             {MENU                          }
             {VARIANT                       }
             {LAYOUT-MANAGER [layout-name] }
```

Syntax Rules

1. The column on the left shows the accepted abbreviations for the terms on the right:

   ```
   COMP      COMPUTATIONAL
   COMP-1    COMPUTATIONAL-1
   COMP-2    COMPUTATIONAL-2
   COMP-3    COMPUTATIONAL-3
   COMP-4    COMPUTATIONAL-4
   COMP-5    COMPUTATIONAL-5
   COMP-6    COMPUTATIONAL-6
   COMP-X    COMPUTATIONAL-X
   COMP-N    COMPUTATIONAL-N
   ```

2. A USAGE clause may be used in any data description entry except those with level-numbers 66, 78, and 88.

3. A USAGE clause may not be used with an external floating-point data item.

4. If a USAGE clause is in the data description entry for a group item, then any USAGE clauses that appear for subordinate entries must be of the same type.

5. The PICTURE string of a COMP, COMP-1, COMP-2, COMP-3, COMP-4, COMP-5, COMP-6, BINARY, or PACKED-DECIMAL item can contain only the symbols “9”, “S”, “V”, and “P”. COMP-6 items may not use the “S” symbol.

6. The PICTURE string of a COMP-X or COMP-N item may contain only all “9” symbols or all “X” symbols.

7. The data description entry for a USAGE IS INDEX data item cannot contain any of the following clauses: BLANK WHEN ZERO, JUSTIFIED, PICTURE, and VALUE IS.

8. Level 88 items may not be specified for a USAGE IS INDEX data item.

9. The data description entry for a USAGE IS POINTER data item cannot contain any of the following clauses: BLANK WHEN ZERO, JUSTIFIED, or PICTURE. A POINTER data item may have a value clause specified for it, but the value must be the word NULL.

10. The data description entry for a USAGE IS FLOAT or a USAGE IS DOUBLE data item cannot contain any of the following clauses: BLANK WHEN ZERO, JUSTIFIED, or PICTURE. FLOAT or DOUBLE data items may have a value clause. The value may be a floating point literal, a numeric literal, or the word ZERO. Here is an example of a Working-Storage Section data item:

```
01  F-DATA-1  USAGE IS FLOAT
              VALUE IS 3.97E+24.
```

11. The following are collectively called the “C-style” data types: SIGNED-INT, UNSIGNED-INT, SIGNED-SHORT, UNSIGNED-SHORT, SIGNED-LONG, UNSIGNED-LONG. These data types are similar to the data types found in the C programming language.

    The data description entry for a C-style data type cannot contain any of the following clauses: BLANK WHEN ZERO, JUSTIFIED, or PICTURE.

12. *Control-type* is one of the graphical control type names known to the compiler, such as LABEL or ENTRY-FIELD, or the name of an ActiveX, COM, or .NET control.

13. The data description entry for USAGE HANDLE data items may not contain any of the following clauses: BLANK WHEN ZERO, JUSTIFIED, or PICTURE. If it contains a VALUE clause, the value specified must be the word NULL.

14. *Font-name* is one of the following identifiers: DEFAULT-FONT, FIXED-FONT, TRADITIONAL-FONT, SMALL-FONT, MEDIUM-FONT, LARGE-FONT.

15. It should be noted that either the "-Df" option or the "-Cv" option will cause the compiler to treat COMP-1 and COMP-2 as FLOAT and DOUBLE, respectively. For more information, see section 5.4 in *Transitioning to ACUCOBOL-GT.*

16. *Layout-name* is the name of one of the system's standard layout managers. Currently, this can only be LM-RESIZE.

General Rules

1. A USAGE clause written at a group level applies to every elementary item subordinate to that group item.

2. If no USAGE clause is specified, then USAGE IS DISPLAY is implied.

3. The internal format of a USAGE IS DISPLAY item is ASCII.

4. The format of an index item is 32-bit signed binary. Its size is always four, and it holds a range of values from -2147483647 to 2147483647. When using a compile switch for compatibility with versions prior to 6.0.0 (-Z52 for example) an index item is 16-bit unsigned binary, size is always two, and it holds values from 0 to 65535.

5. The format of a COMP-1 data item is 16-bit signed binary. The legal values range from -32767 to 32767. The size of the data item is always two bytes, and the high-order half of the data is stored in the leftmost byte. The PICTURE string that describes the item is irrelevant. Unlike other numeric data types, a size error will occur on a COMP-1,

COMP-X, or COMP-N data item only when the value exceeds the physical storage of the item (in other words, the number of "9"s in the item's PICTURE is ignored when size error is determined).

6. For COMP-2 (decimal storage), each digit is stored in one byte in decimal format. If the value is signed, then an additional trailing byte is allocated for the sign. The storage of COMP-2 is identical with USAGE DISPLAY with the high-order four bits stripped from each byte.

7. For COMP-3 (packed-decimal storage), two digits are stored in each byte. An additional half byte is allocated for the sign, even if the value is unsigned. The sign is placed in the rightmost position, and its value is 0x0D for negative; all other values are treated as positive (but see rule 18 below). The size of an item (including one for the implied sign) is divided by two to arrive at its actual size (rounding fractions up).

8. The format of a COMP-4 item is two's-complement binary (the value without its decimal point). COMP-4 values are stored in a machine-independent format. This format places the highest-order part of the value in the leftmost position and follows down to the low-order part in the rightmost position. The number of bytes a data item occupies depends on the number of "9"s in its PICTURE and on the presence of various compile-time options. For example, you may include more than eighteen "9"s only if your program has been compiled for 31-digit support. This is summarized in the following table:

# of "9"s	Default	-D1	-Dm	-D7
1-2	2	1	1	1
3-4	2	2	2	2
5-6	4	4	3	3
7	4	4	4	3
8-9	4	4	4	4
10-11	8	8	5	5
12	8	8	6	5
13-14	8	8	6	6

# of "9"s	Default	-D1	-Dm	-D7
15-16	8	8	7	7
17-18	8	8	8	8
19	12	12	9	8,9
20	12	12	9	9
21	12	12	9	9
22	12	12	10	10
23	12	12	10	10
24	12	12	11	10,11
25	12	12	11	11
26	12	12	11	11
27	12	12	12	12
28	12	12	12	12
29	16	16	13	13
30	16	16	13	13
31	16	16	13	13

Note: Where two values are given, the smaller value applies to unsigned data items, and the larger value applies to signed data items.

9. COMP-5 is primarily used to communicate with external programs that expect native data storage.

   The format of a COMP-5 data item is identical to a COMP-4 data item, except that the data is stored in a machine-dependent format. It is stored in an order that is natural to the host machine. For example, a PIC S9(9) COMP-5 data item is equivalent to a 32-bit binary word on the host machine, and a PIC S9(20) COMP-5 item is equivalent to a 64-bit word.

Note: Data stored in a COMP-5 field may not be transportable to other machines because different machines have different natural byte-orderings. On many machines (68000, most RISC), COMP-5 is identical to COMP-4. On others (80x86, VAX), it is the same with the bytes in the reverse order.

A VALUE clause for a COMP-5 data item is stored in a machine-independent format and is adjusted when it is loaded into the data item. This ensures that the value is the same from machine to machine.

On arithmetic and non-arithmetic stores into COMP-5 items, if truncation is required, by default ACUCOBOL-GT truncates in decimal to the number of digits given in the PICTURE clause. You can use the "--TruncANSI" compiler option to force truncation in binary to the capacity of the allocated storage of COMP-5 items. The "-Dz" and "--noTrunc"options also affect truncation. See Book 1, section 2.1.9, "Data Storage Options," for more information.

Level 01 and level 77 data items that are COMP-5 are automatically synchronized to an appropriate machine boundary, regardless of any compile-time settings. This allows you to pass these items safely to C subroutines without having to concern yourself with alignment.

If COMP-5 is used with a PIC X(n) data item and assigned an alphanumeric value, the results are undefined. For example, the following code fragment causes NUM to have an undefined number and the resulting value for the last line will be "100":

```
NUM PIC X(5) COMP-5.
ALPHANUM PIC X(9).
MOVE "ABC" TO NUM.
MOVE "1,000" TO NUM.
MOVE ALPHANUM TO NUM.
MOVE "100" TO NUM.
```

A PIC X(n) data item used with COMP-5 cannot be signed.

10. The format of a COMP-6 item is identical to a COMP-3 item except that it is unsigned and no space is allocated for the sign. If the number of digits is odd, a zero is added to the left end of the number before it

is packed. Thus there are two decimal digits per byte, and the actual size of the item is determined by dividing its PICTURE size by two and rounding up.

11. A COMP-X data item must be described with a picture string consisting of only "9" or only "X" symbols. In either case, the data item is treated as an unsigned binary integer, with internal storage similar to that of a COMP-4 data item. If "X" symbols are used to describe the item, then the number of bytes allocated to the item is the same as the number of "X" symbols in the picture string. If "9" symbols are used instead, then the number of bytes allocated is the least number of bytes required to hold a number of that size. For example, a "PIC 99" data item will be allocated 1 byte; a "PIC 9(9)" data item will be allocated 4 bytes.

    Regardless of the number of "9" symbols in the item's picture string, the maximum value that can be stored in a COMP-X item is determined by the number of bytes allocated to it (to a maximum of 18 digits, or a maximum of 31 digits if 31-digit support is in effect). For example, a COMP-X item consisting of 1 byte can hold a range of numbers from 0 to 255. A 2-byte COMP-X number can hold from 0 to 65535. A size error occurs on a COMP-X item only when the value is larger than the data item can physically hold. When COMP-X is used with a PIC(X) data item, the maximum is PIC X(8). (This maximum is increased to PIC X(16) when 31-digit support is in effect.)

12. A COMP-N data item is identical to a COMP-X data item, except that the data is stored in the host machine's native format, instead of machine-independent format.

13. Data items described as PACKED-DECIMAL are identical to COMP-3. You can cause unsigned PACKED-DECIMAL to be treated as COMP-6 by using a compile-time option.

14. By default, a BINARY data item is identical to a COMP-4 data item. The compile-time option "-D5" treats BINARY data items as COMP-5 items instead.

15. In VAX/COBOL compatibility mode, a COMP data item is the same as COMP-4 and is treated as binary data. In RM/COBOL compatibility mode, COMP is the same as COMP-2. You can use compile-time options to change the default behavior.

16. A pointer data item is treated as an unsigned numeric data item. The internal format differs for each machine. Pointer data items are intended to hold addresses of other data items (see the **SET Statement**.) A pointer data item may have a VALUE clause specified for it, but the specified value must be the word NULL. This indicates that the pointer does not currently point to any item. If a pointer is not explicitly given an initial value, then its initial value is arbitrary.

    Pointer data items occupy 8 bytes. This provides enough space to hold an address on a 64-bit machine. If you are on a smaller machine, the runtime uses only the first 4 bytes of pointer data items (the trailing 4 bytes remain in memory, they are just left unused). You can do this to conserve storage if you know you will not be running on a 64-bit machine.

    Pointers may be used in conditional expressions, where they can be compared to each other or to the value NULL. A comparison involving a pointer must be either “equals” or “not equals” (“greater” and “less than” comparisons are not allowed).

    Level 01 and level 77 data items that are POINTER items are automatically synchronized to an appropriate machine boundary, regardless of any compile-time settings. This allows you to pass these items safely to C subroutines without having to concern yourself with alignment.

    Except for the automatic synchronization, USAGE POINTER data items are treated in all respects like USAGE UNSIGNED-LONG data items. This handles all current machines correctly. This behavior may change to meet the requirements of some future machine.

17. Floating-point data items are stored in a machine-dependent format. USAGE FLOAT items have 4 bytes allocated to them. USAGE DOUBLE items occupy 8 bytes.

    Level 01 and level 77 data items that are USAGE FLOAT or DOUBLE are automatically synchronized to appropriate machine boundaries, regardless of any compile-time settings. This allows you to pass these items safely to C subroutines without having to concern yourself with alignment.

18. The ANSI definition of COBOL does not state how signs should be stored in numeric fields (except for the case of SIGN IS SEPARATE). ACUCOBOL-GT lets you select alternate sign-storage conventions by using the compile-time options “-Dca”, ‘-Dcb”, “-Dci”, “-Dcm”, “-Dcn”, “-Dcr”, and “-Dcv”. Specifying a sign-storage convention is sometimes useful when you are exporting and importing data. For additional information, see the *User’s Guide*, **section 2.2.10, “Data Storage Options.”**

    The storage convention affects how data appears in USAGE DISPLAY, COMP-2, and COMP-3 data types. In USAGE DISPLAY, standard ASCII storage, if the sign is incorporated into a digit position, the digit is encoded according to the following table:

    USAGE DISPLAY

DIGIT VALUE	**-Dca, -Dcb, -Dcm, -Dcr Positive**	**-Dci, -Dcn Positive**	**-Dca, -Dci, -Dcn Negative**	**-Dcb Negative**	**-Dcm Negative**	**-Dcr Negative**
0	‘0’	‘{‘	‘}’	‘@’	‘p’	‘ ’ (space)
1	‘1’	‘A’	‘J’	‘A’	‘q’	‘!’
2	‘2’	‘B’	‘K’	‘B’	‘r’	‘”’ (double-quote)
3	‘3’	‘C’	‘L’	‘C’	‘s’	‘#’
4	‘4’	‘D’	‘M’	‘D’	‘t’	‘$’
5	‘5’	‘E’	‘N’	‘E’	‘u’	‘%’
6	‘6’	‘F’	‘O’	‘F’	‘v’	‘&’
7	‘7’	‘G’	‘P’	‘G’	‘w’	‘’’ (single-quote)
8	‘8’	‘H’	‘Q’	‘H’	‘x’	‘(‘
9	‘9’	‘I’	‘R’	‘I’	‘y’	‘)’

Note: For import compatibility with some systems that do not have the symbols "{" and "}", the symbols "[" and "?" are considered equivalent to "{", and the symbols "]", ":", and "!" are considered equivalent to "}", when an item with USAGE DISPLAY is read.

The next two tables show sign representation for COMP-2 and COMP-3 items, when you are using the "-Dca", "-Dcb", "-Dci", "-Dcm", "-Dcn", "-Dcr", and "-Dcv" storage conventions. For COMP-2, the trailing byte is reserved for the sign. For COMP-3, the trailing half-byte is reserved for the sign.

USAGE COMP-2

-Dca Positive	x'0B'
-Dcb/-Dci/-Dcm/-Dcn/-Dcr Positive	x'0C'
-Dca/-Dcb/-Dci/-Dcm/-Dcn/-Dcr Negative	x'0D'

USAGE COMP-3

-Dca Positive	x'0F'
-Dcb/-Dci/-Dcm/-Dcr Positive	x'0C'
-Dca/-Dcb/-Dci/-Dcm/-Dcr Negative	x'0D
-Dca/-Dcb/-Dci/-Dcm/-Dcr Unsigned	x'0F'
-Dcv Unsigned	x'0C'

19. There are six USAGE types for integer data that simplify communications with other programming languages such as C. These types are designed to provide a portable method for handling machine-dependent data. The six USAGE types handle three classes of machine data: "short words," "words," and "long words." These three correspond to the C data types: "short", "int", and "long". There are signed and unsigned versions of each of these data types.

    These USAGE types are specified without a PICTURE clause (like USAGE INDEX and POINTER).

The names of the types are:

SIGNED-SHORT	UNSIGNED-SHORT
SIGNED-INT	UNSIGNED-INT
SIGNED-LONG	UNSIGNED-LONG

Each of these represents a binary value that is stored using the machine's native byte ordering. Since there is no PICTURE phrase, size checking for these items is performed only on byte boundaries. These data types are automatically SYNCHRONIZED.

The unusual characteristic of these data types is that their size is not necessarily set at compile time. Instead, the size of these items is determined at execution time. This allows them to match the working characteristics of the host machine. For example, a SIGNED-LONG data item will contain 64 bits when run on a DEC Alpha machine, but it will have 32 bits when run on an Intel 80486-based machine. This lets you write one program that can communicate effectively with an external routine written in another language (such as C), regardless of the target environment.

In order to lay out memory, the compiler assigns a maximum size to each of these data types. This is the number of bytes that the item will occupy. At run time, these items may be reduced in size to match the host machine's characteristics. Any remaining bytes are then treated as FILLER. The "-Dw" compile option (see the *User's Guide* **section 2.2.10, "Data Storage Options."**) determines the maximum size of these types:

USAGE	**-Dw32**	**-Dw64**
SIGNED-SHORT UNSIGNED-SHORT	2*	2*
SIGNED-INT UNSIGNED-INT	4	4
SIGNED-LONG UNSIGNED-LONG	4*	8

Table entries marked with an asterisk indicate fixed-size items. A fixed-size item is the same size regardless of the target machine. Entries without an asterisk are variable in size. These items will occupy space up to the number of bytes listed in the table.

Note: The sizes listed in the table above cover all current and anticipated machines that run ACUCOBOL-GT. Future architectures may require changes to the maximum size assigned to these items.

In the execution environment, these items act in all ways as if they were fixed-size data items of the appropriate size.

For example, the following code fragment:

```
77  LONG-1         SIGNED-LONG.
77  SIZE-1         PIC 9.

SET SIZE-1 TO SIZE OF LONG-1.
DISPLAY SIZE-1.
```

will print “4” when run on a 32-bit machine, but it will print “8” when run on a 64-bit machine.

Examples

In the following examples, each byte is represented by two hexadecimal digits or by a single quoted character. Each value is shown in the various formats. Also shown is USAGE DISPLAY using the various SIGN options. The following examples use the default ACUCOBOL-GT sign-storage conventions.

PIC 9(3) VALUE 123.

```
TRAILING            '1'   '2'   '3'
TRAILING SEPARATE   '1'   '2'   '3'
LEADING             '1'   '2'   '3'
LEADING SEPARATE    '1'   '2'   '3'
COMP-1                    00    7B
COMP-2              01    02    03
COMP-3                    12    3F
COMP-4                    00    7B
COMP-5(68000)             00    7B
COMP-5(8086)              7B    00
```

```
COMP-6                    01    23
```

PIC S9(3) VALUE -123.

```
TRAILING                 '1'   '2'   'L'
TRAILING SEPARATE  '1'   '2'   '3'   '-'
LEADING                  'J'   '2'   '3'
LEADING SEPARATE   '-'   '1'   '2'   '3'
COMP-1                         FF    85
COMP-2             01    02    03    0D
COMP-3                         12    3D
COMP-4                         FF    85
COMP-5(68000)                  FF    85
COMP-5(8086)                   85    FF
COMP-6             illegal
```

PIC 9(5)V9 VALUE 12345.6.

```
TRAILING             '1'   '2'   '3'   '4'   '5'   '6'
TRAILING SEPARATE    '1'   '2'   '3'   '4'   '5'   '6'
LEADING              '1'   '2'   '3'   '4'   '5'   '6'
LEADING SEPARATE     '1'   '2'   '3'   '4'   '5'   '6'
COMP-1               illegal
COMP-2               01    02    03    04    05    06
COMP-3                           01    23    45    6F
COMP-4                           00    01    E2    40
COMP-5(68000)                    00    01    E2    40
COMP-5(8086)                     40    E2    01    00
COMP-6                                 12    34    56
```

PIC S9(5)V9 VALUE -12345.6.

```
TRAILING                 '1'  '2'  '3'  '4'  '5'  '0'
TRAILING SEPARATE  '1'   '2'  '3'  '4'  '5'  '6'  '-'
LEADING                  'J'  '2'  '3'  '4'  '5'  '6'
LEADING SEPARATE   '-'   '1'  '2'  '3'  '4'  '5'  '6'
COMP-1             illegal
COMP-2             01   02   03   04   05   06   0D
COMP-3                          01   23   45   6D
COMP-4                          FF   FE   1D   C0
COMP-5(68000)                   FF   FE   1D   C0
COMP-5(8086)                    C0   1D   FE   FF
```

```
COMP-6              illegal
```

20. HANDLE data items make up their own data class and category in COBOL. Internally they are stored as integer values, and behave like numbers when used. A HANDLE data item is normally used to store the handle of a dynamically created object such as a floating window or a graphical control.

    HANDLE data items come in two forms: *typed* and *generic*. You create a generic handle when you omit the OF phrase. You create a typed handle when you include the OF phrase.

21. You may use HANDLE data items only when explicitly allowed, or as part of a MOVE statement, a CALL statement (as a parameter), or in a Boolean expression.

22. Generic handles may be used in any situation where handles are allowed. When you use a generic handle as the source of a MODIFY statement, you will not be able to use any control-specific property or style names in that statement. This is because the generic handle could be associated with any type of control. In this case, the compiler cannot determine which set of style and property names is valid.

23. Typed handles may be used in statements where any handle is allowed, or when you are referring to an object of a matching type. For example, a HANDLE OF WINDOW cannot be used as the handle in a DISPLAY LABEL statement. Instead, you must use either a generic handle or a HANDLE OF LABEL. Typed handles allow the compiler to recognize associated style and property names when appropriate. Typed handles also improve the readability of your program by providing additional information about the intended use of the handle, in addition to providing compile-time checking to ensure that you are using the handles in appropriate situations.

24. Handles may be used in comparisons. There are only two meaningful comparisons: checking for equality or inequality to NULL, and comparison to another handle data item. A handle value of NULL always indicates an invalid handle.

25. Handles are stored internally as 4-byte binary integers. This information can be useful when you are debugging a program (you can examine the values of handles in the debugger). You should not rely on this definition in your program, however, because it is subject to change in the future.

26. Handle data items are automatically SYNCHRONIZED on a 4-byte boundary. Note that this occurs regardless of the setting of the "-Dl" compile-time option (which limits the amount of synchronization). The runtime system requires this level of alignment to avoid generating bus errors on some machines.

27. If *font-name* is specified, then the data item described by the USAGE clause is initialized at program startup with the corresponding font handle. This acts identically to placing the statement:

    ```
    ACCEPT data-item FROM STANDARD OBJECT "font-name"
    ```

    at the beginning of your program, where *data-item* is the data item described by the USAGE clause and *font-name* is the same as *font-name* in the USAGE clause.

5.7.1.9 SIGN clause

The SIGN clause specifies the location and format of an item's operational sign.

General Format

```
[ SIGN IS ]  {LEADING } [ SEPARATE CHARACTER ]
             {TRAILING}
```

Syntax Rules

1. The SIGN clause may appear only in a numeric data description entry whose PICTURE string contains the "S" symbol, or on a group item that contains such an entry.

2. The item must have USAGE DISPLAY.

3. A SIGN clause may not be used with an external floating-point data item.

General Rules

1. If the SIGN clause is omitted for a data item, then the operational sign is incorporated into the final digit of the data item. The “S” symbol does not occupy any space in the data item in this case.
2. If the SIGN clause is specified, but without the SEPARATE CHARACTER phrase, then the sign is incorporated into the first or last digit of the data item as specified.
3. If the SEPARATE CHARACTER phrase is specified in the SIGN clause, then the “S” symbol represents a separate character position and it adds one to the size of the data item. This character is located as the first or last character of the data item as specified in the clause. The sign is represented with a “+” or “-” character as appropriate. The zero value may have either sign.
4. If the SIGN phrase is specified for a group item, then it applies to each subordinate elementary numeric data item. A SIGN phrase specified for an elementary data item takes precedence over a SIGN phrase specified for one of its group items. If more than one group item has a SIGN phrase specified for it in a hierarchy, the lowest level one takes precedence.

5.7.1.10 OCCURS clause

The OCCURS clause allows for the creation of tables or arrays.

General Format

Format 1

```
OCCURS table-size TIMES

   [ {ASCENDING } KEY IS {key-name} ... ] ...
     {DESCENDING}

   [ INDEXED BY {index-name} ... ]
```

Format 2

```
OCCURS [ min-size TO ] max-size TIMES DEPENDING ON dep-item

   [ {ASCENDING } KEY IS {key-name} ... ] ...
```

```
    {DESCENDING}

  [ INDEXED BY {index-name} ... ]
```

Syntax Rules

1. *Table-size* is a positive integer that specifies the exact number of occurrences.
2. *Min-size* is an integer that specifies the minimum number of occurrences. Its value must be greater than or equal to zero. If omitted, it has a default value of one.
3. *Max-size* is an integer that specifies the maximum number of occurrences. Its value cannot be less than *min-size*; it cannot be larger than 2147483647.
4. *Dep-item* is the name of an elementary unsigned integer data item. Its value specifies the current number of occurrences.
5.
6. *Key-name* is the data-name of the entry that contains the OCCURS clause, or an entry subordinate to it. *Key-name* may be qualified. Each *key-name* after the first must name a subordinate item to the entry containing the OCCURS clause. A *key-name* may not contain an OCCURS clause unless that *key-name* is the name of the subject of the current OCCURS clause.
7. If a Format 2 OCCURS clause appears in a record description for a file, *dep-item* must appear in the same record.
8. An item described by a Format 2 OCCURS clause can be followed, in the same record description, only by items subordinate to it.
9. An OCCURS clause may not appear in a data description entry that has a level-number of 66, 78, or 88. A variable occurrence data item (one that has a Format 2 OCCURS clause) may not be subordinate to another data item that has an OCCURS clause.
10. The *dep-item* may not occupy any character positions subordinate to the data item described by that OCCURS clause.

General Rules

1. The OCCURS clause defines tables. All items subordinate to an OCCURS clause must be referenced with subscripting or indexing.
2. Except for the OCCURS clause itself, all data description clauses associated with that data item apply to each occurrence of the item.
3. A Format 1 OCCURS clause defines a fixed-size table.
4. A Format 2 OCCURS clause defines a table that contains a variable number of occurrences. The current number of occurrences depends on *dep-item*. Only the number of occurrences is variable; the table's size is fixed (*max-size*). *Dep-item* must fall in the range from *min-size* to *max-size*. Values contained in the table that are beyond the current number of occurrences (*dep-item*) are unpredictable.
5. The size occupied by a table is the size of one of its occurrences multiplied by the maximum number of occurrences in the table (*max-size* in Format 2).
6. If a group item containing a subordinate data item with a Format 2 OCCURS clause is involved in an operation, the operation uses only that part of the table specified by *dep-item*.
7. Index names are treated in all respects as USAGE INDEX data items by the compiler.
8. The KEY IS phrase indicates that the data is arranged in ascending or descending order according to the values in the data items named by *key-name*. The position of each *key-name* in the list determines its significance. The first *key-name* is the most significant, the last is the least significant. The KEY IS phrase is used by the SEARCH ALL verb.

5.7.1.11 SYNCHRONIZED clause

The SYNCHRONIZED clause specifies elementary item alignment on word boundaries of the computer's memory.

General Format

```
{SYNCHRONIZED}  [LEFT ]
{SYNC         }  [RIGHT]
```

Syntax Rules

1. SYNC is an abbreviation of SYNCHRONIZED.
2. The SYNCHRONIZED clause can be used for elementary items only.
3. A SYNCHRONIZED clause may not be used with an external floating-point data item.

General Rules

1. The SYNCHRONIZED clause is used to specify that word boundary alignment should be performed for the data item. Normally, data contained in records is aligned on byte boundaries. Only data items whose underlying representation is binary are affected by the SYNCHRONIZED clause.
2. The SYNCHRONIZED clause causes the data item to be placed on a boundary that is an even multiple of the natural size of the data item. The following table lists the boundary used for each size of data item:

Data Size	**Boundary Multiple**
1-2	2
3-4	4
5-8	8

3. A group item that contains a synchronized data item is also synchronized on the same boundary. Regardless of the effects of synchronization, a group item always begins at the same location as its first elementary data item.
4. Synchronization may result in the creation of filler bytes. These bytes count in the size of any group item that contains them. For this reason, a group item that contains synchronized data may be larger than the total size of its elementary items.
5. Level 01 and level 77 data items that are not otherwise synchronized are placed on a boundary that can be selected at compile time. By default, these items are placed at word boundaries that are divisible by two.

6. Level 01 and level 77 data items that are POINTER or COMP-5 items are automatically synchronized to an appropriate machine boundary, regardless of any compile-time settings. All C-style data types are automatically synchronized regardless of their level. This allows you to pass these items safely to C subroutines without having to concern yourself with alignment.

7. A compile-time ("-Dl") option can be used to cut back the maximum boundary multiple. For example, "-Dl4" would cause items of size 1 or 2 to be synchronized on 2-byte boundaries, and all other items synchronized on 4-byte boundaries. If this option is not specified, then the maximum boundary multiple depends on the compatibility mode being used:

Mode	Boundary Limit
VAX COBOL	8
RM/COBOL	2
ICOBOL	1

   A limit of 1 effectively inhibits synchronization.

8. The LEFT and RIGHT options are treated as commentary. They have the same effect as a SYNCHRONIZED clause without either option.

5.7.1.12 JUSTIFIED clause

The JUSTIFIED clause specifies alternate data positioning rules for alphanumeric data.

General Format

```
{JUSTIFIED} RIGHT
{JUST     }
```

Syntax Rules

1. JUST is an abbreviation for JUSTIFIED.

2. The JUSTIFIED clause may be used for elementary items only.

3. The JUSTIFIED clause may not be used on index, numeric, or edited data items. It may be used only on alphabetic and alphanumeric data items.

General Rules

When an operation transfers data to an item described with the JUSTIFIED clause, the standard alignment rules are altered.

1. If the sending item is larger than the receiving item, excess characters are truncated from the left.
2. If the sending item is smaller than the receiving item, data is aligned at the rightmost character position of the receiving item. The excess characters on the left are filled with spaces.

5.7.1.13 BLANK WHEN ZERO clause

The BLANK WHEN ZERO clause causes the data item to be filled with spaces when its value is zero.

General Format

```
BLANK WHEN ZERO
```

Syntax Rules

1. The BLANK WHEN ZERO clause may be used only for a numeric or numeric edited elementary item whose picture does not contain “S” or “*”.
2. The data item must have USAGE DISPLAY.

Note: The compiler accepts any sign designation other than “S” without declaring an error. That includes “+”, “-”, “CR”, and “DB”.

General Rules

1. The BLANK WHEN ZERO clause causes an item to be filled with spaces when its value is zero.

2. Any numeric item described with a BLANK WHEN ZERO clause becomes a numeric edited item.

5.7.1.14 VALUE clause

The VALUE clause defines the initial value of Working-Storage. It also describes the values associated with conditionals.

General Format

Format 1

```
VALUE IS value-lit
```

Format 2

```
{VALUE IS  } { low-val [ {THROUGH} high-val ] } ...
{VALUES ARE}             {THRU   }

    [ WHEN SET TO FALSE false-val ]
```

Format 3

```
78 user-name VALUE IS {literal-1} [ {+} literal-2 ] .
                      {NEXT     }   {-}
                                    {*}
                                    {/}
```

Syntax Rules

1. *Value-lit* is a numeric or non-numeric literal that defines the initial value of a Working-Storage item.

2. *Low-val* is a numeric or non-numeric literal that defines the value of a condition, or the lower value of a condition range.

3. *High-val* is a numeric or non-numeric literal that defines the upper value of a condition range. It must be the same type as *low-val* and must have a value greater than *low-val*.

4. *False-val* is a numeric or non-numeric literal that defines the FALSE value for the corresponding data item.

5. *Literal-1* is a numeric or alphanumeric literal. If *literal-2* is specified, then *literal-1* must be a numeric, non-floating-point literal. *Literal-1* can also be a "LENGTH OF" expression, as described in **section 2.1.2.1, "Numeric literals."**

6. *Literal-2* is a numeric, non-floating point literal or a "LENGTH OF" expression.

7. The VALUE clause may not be used for any item whose size is variable.

8. A VALUE clause may not be used with an external floating-point data item.

9. All literals used in a VALUE clause must have a value which falls within the range of allowed values for the item's PICTURE clause. Non-numeric literals may not exceed the size of the item. Numeric items must have numeric literals. Alphabetic, alphanumeric, group, and edited items must have non-numeric literals.

10. The words THROUGH and THRU are equivalent.

11. The Format 2 VALUE clause may be used only in a condition-name (level 88). Its use is required in this case.

12. VALUE clauses may appear in the File Section and the Linkage Section. They have no effect in these sections unless they are part of condition-name entries (level 88s) or named constants (level 78s).

    Their presence in these two sections simplifies the management of COPY libraries. For example, if you plan to use the same COPY library in Working Storage in program-A and in Linkage in program-B, you need not remove the VALUE clauses in the Linkage Section.

13. The VALUE clause may not be specified for a group item that contains subordinate items with any of the following clauses: JUSTIFIED, SYNCHRONIZED, or USAGE (other than USAGE DISPLAY).

14. A Format 1 VALUE clause may not appear on a data item that is subordinate to a REDEFINES clause.

15. A level 78 entry associates a value with the name of a constant, and *user-name* is a user-defined word that names the constant. *User-name* must be unique, because it may not be qualified.

General Rules

1. A Format 1 VALUE clause specifies the initial state of a Working-Storage item or the value of a named constant. A Format 2 VALUE clause defines a condition-name. A Format 3 VALUE clause defines a constant.

2. When a VALUE clause is applied to an edited item, that item is treated as if it were alphanumeric. Editing characters in the PICTURE clause count toward the size of the item but have no effect on initialization. The literals must therefore appear in edited form.

Initialization (Format 1)

1. A Format 1 VALUE clause takes effect only when the program enters its initial state.

2. The VALUE clause initializes its data item to the value of *value-lit*.

3. If no VALUE clause is specified, the initial value of a Working-Storage item is set to spaces, or the value specified with the "-Dv" compile option. This may, or may not, be a legal value for the item.

4. When a VALUE clause appears on a data item that is subordinate to an x OCCURS clause, every occurrence of that data item is initialized to the specified value.

5. When a VALUE clause is applied to a group item, that item is initialized as if it were an alphanumeric item. It is not affected by characteristics of any subordinate items to the group. No subordinate item may contain a VALUE clause within this group.

6. The BLANK WHEN ZERO and JUSTIFIED clauses do not affect initialization.

Condition-Name (Format 2)

1. The VALUE clause is required in a condition-name entry. The only clauses allowed in a condition-name entry are the level-number (88), the condition-name itself, and its VALUE clause. See **section 5.2.4** for examples of condition-name entries.

2. The characteristics of the condition-name are implicitly the same as those of its condition-variable. The condition-variable is the immediately preceding completed record description entry.

3. The VALUE clause describes the values of the condition-variable that imply a "true" state for the associated condition-name. This consists of a single value, a range of values, or a set of both single values and ranges. For example "VALUES ARE 1, 2, 4 THRU 7" would define a condition-name that was "true" when its associated condition-variable had any of the values "1", "2", "4", "5", "6", and "7".

4. The WHEN SET TO FALSE phrase defines the "false" value for the condition-name. The SET statement cannot set the condition-name to FALSE unless a "false" value is specified here.

Level 78 Constant (Format 3)

1. When it is used with a level 78 item, the VALUE clause associates a literal with a user-defined word. The user-defined word is then called a *named constant.* A named constant may be used anywhere the corresponding literal may be used. The compiler replaces each occurrence of the named constant with the literal.

2. The literal is constructed as follows:

   a. If *literal-1* is specified (without *literal-2*), then *user-name* acts as a synonym for that literal in the remainder of the program.

   b. If NEXT is specified (without *literal-2*), then *user-name* acts as an integer numeric constant whose value is the virtual address of the first byte past the end of the immediately preceding data item. However, if the immediately preceding data item is a group item, then the value is the virtual address of the beginning of the group item. Note that the effect of synchronization and data alignment may mean that the next data item does not start at the same virtual address as the first byte past the end of the previous data item. This construct has undefined effects if the immediately preceding data item is, or is part of, a data item greater than 64KB in size.

Caution: The use of NEXT is designed for compatibility with other COBOL compilers. The effects of data alignment and data space segmentation make this feature difficult to use with standard ACUCOBOL-GT code. We do not recommend its use except when you are migrating code that already contains similar syntax. ACUCOBOL-GT provides other techniques for address manipulation (e.g. POINTER data items) and size computation (e.g. SET TO SIZE OF statement).

c. If *literal-2* is specified, then *user-name* is an integer numeric constant whose value is the same as it would be without *literal-2* specified, acted upon by the specified operation. For example, the following two level 78s have the same value:

```
78  THREE          VALUE 3.
78  THREE-AGAIN    VALUE 1 + 2.
```

In some cases, *literal-1* and *literal-2* may, themselves, be level 78s. For example:

```
78  ONE            VALUE 1.
78  TWO            VALUE ONE + 1.
78  THREE          VALUE TWO + 1.
```

When *literal-2* is used, both *literal-1* and *literal-2* are evaluated as integers, and the arithmetic is done using 32-bit integer arithmetic. The result is always an integer.

3. You may use a level 78 named constant as a repeat count in a PICTURE string. This means that, in a PICTURE string, you may substitute a level 78 for a number in parentheses. In the following example, DATA-1 and DATA-2 are both the same size:

```
78  LENG-20    VALUE 20.
01  DATA-1     PIC X(20).
01  DATA-2     PIC X(LENG-20).
```

5.7.1.15 RENAMES clause

The RENAMES clause provides an alternate data name for a set of data items.

General Format

```
66 new-name RENAMES rename-1 [ {THRU   } rename-2 ]
                               {THROUGH}
```

Syntax Rules

1. The level-number (66) and *new-name* are not actually part of the RENAMES clause. They are included in the General Format for clarity.
2. *New-name* is a user-defined word that is the name of the item being described.
3. *Rename-1* is the data name of the leftmost data item in the area.
4. *Rename-2* is the data name of the rightmost data item in the area.
5. All RENAMES entries referring to data items in a logical record must immediately follow the last data description entry of that record.
6. *New-name* may not be used as a qualifier.
7. *New-name* must be unique, because it may not be qualified.
8. *Rename-1* and *rename-2* must be the names of items in the same logical record.
9. The data description entries for *rename-1* and *rename-2* may not contain or be subordinate to an OCCURS clause.
10. A level 66 entry may not rename another level 66 entry. Nor can it rename a level 78, 88, level 01, or level 77 entry.
11. None of the items in the range from *rename-1* to *rename-2* may contain variable occurrence items.
12. The words THRU and THROUGH are equivalent.
13. *Rename-2* may not be subordinate to *rename-1*. The beginning of *rename-2* cannot be to the left of the beginning of *rename-1*. The end of *rename-2* must be to the right of the end of *rename-1*.

General Rules

1. If *rename-2* is used, *new-name* contains all the character positions between the start of *rename-1* and the end of *rename-2*.

2. If *rename-2* is used, *new-name* is treated as a group item. If *rename-2* is not used, all the data attributes for *rename-1* become attributes of *new-name*. In this case, you are providing an alternate name for a single data item.

5.8 Screen Section

The Screen Section describes the format, layout, and behavior of console screens. It contains one or more uniquely named screen description entries.

A screen item in the Screen Section is analogous to a data item in the Working-Storage Section. The terms “screen item” or “Screen Section entry” may refer to an elementary item or a group item.

Screen Section entries:

- provide “form level” ACCEPT and DISPLAY, which means a single ACCEPT or DISPLAY statement can activate any number of elements on the screen.
- are referenced by Format 2 ACCEPT and DISPLAY statements in the Procedure Division.
- define one or more controls, data fields, or literals, and their associated attributes.

ACCEPT and DISPLAY statements that reference a Screen Section entry include built-in features that greatly simplify programming. A single ACCEPT statement will reference the controls and input fields that you named and defined in a Screen Section entry, and a single DISPLAY statement will display the corresponding collection of prompts for the user. When you code your programs with Format 2 ACCEPT and DISPLAY, the tab and arrow keys (or their substitutes if you have customized the keyboard) are automatically functional so the user can move from field to field during data entry.

See the *User’s Guide* **Section 6.5** for an extended discussion of the Screen Section.

Note: Since Screen Section entries are defined in the Data Division but acted upon in the Procedure Division, you will want to read the information given here with that for the Format 2 ACCEPT and DISPLAY described in **section 6.6**.

General Format

```
SCREEN SECTION.

   [ screen-description ] ...
```

Syntax Rules

1. Each level 01 screen item described in the Screen Section must be uniquely named.
2. Subordinate screen names need not be unique if they can be made unique through qualification.

5.9 Screen Description Entry

A screen description entry specifies the characteristics of a single screen item. Many of the phrases permitted in a Screen Description Entry are explained in **section 6.4.9, "Common Screen Options."**

General Format

Format 1

```
level-number  [ screen-name ]
              [ FILLER      ]
```

Remaining phrases are optional, can appear in any order.

```
GRAPHICAL
CHARACTER

{PICTURE} IS picture-string
{PIC    }
```

```
[ [FROM from-item] [TO to-item] ]
[       USING using-item        ]

[USAGE IS] DISPLAY

[SIGN IS] {LEADING } SEPARATE CHARACTER
          {TRAILING}

OCCURS table-size TIMES

{JUSTIFIED} RIGHT
{JUST     }

BLANK WHEN ZERO

VALUE IS value-lit

{BLANK} {SCREEN}

{ERASE} {LINE  }
        {EOS   }
        {EOL   }

LINE [ NUMBER IS [PLUS] line-no ]
                 [ +  ]
                 [ -  ]

{COLUMN  } [ NUMBER IS [PLUS] col-no ]
{COL     }             [ +  ]
{POSITION}             [ -  ]
{POS     }

SIZE IS length

COLOR IS color-val
COLOUR

FOREGROUND-COLOR IS fg-color
FOREGROUND-COLOUR

BACKGROUND-COLOR IS bg-color
BACKGROUND-COLOUR

{BACKGROUND-HIGH     }
{BACKGROUND-LOW      }
```

```
{BACKGROUND-STANDARD }

{BELL}
{BEEP}

{UNDERLINED}
{UNDERLINE }

{HIGHLIGHT}
{HIGH     }
{BOLD     }
{LOWLIGHT }
{LOW      }
{STANDARD }

{BLINKING}
{BLINK   }

{REVERSE-VIDEO}
{REVERSED     }
{REVERSE      }

SAME

OUTPUT {LEFT    }
       {RIGHT   }
       {CENTERED}

{NO-ECHO}
{NO ECHO}
{SECURE }
{OFF    }

PROMPT [ CHARACTER IS prompt-lit ]

{UPPER}
{LOWER}

{AUTO         }
{AUTO-SKIP    }
{AUTOTERMINATE}
{TAB          }

{REQUIRED   }
{EMPTY-CHECK}
```

```
{FULL        }
{LENGTH-CHECK}

{ZERO-FILL   }
{NUMERIC-FILL}

HELP-ID {IS} help-id
        {= }

ENABLED {IS} enabled-state
        {= }

VISIBLE {IS} visible-state
        {= }

{BEFORE } PROCEDURE IS { proc-1 [{THROUGH} proc-2] }
{AFTER  }                       {THRU   }

{EXCEPTION}            { NULL                      }
```

Format 2

```
level-number [ screen-name ]
             [ FILLER      ]

{control-type-name}
{OBJECT control-type}

[ title ]
```

Remaining phrases are optional, can appear in any order.

```
GRAPHICAL
CHARACTER

{IDENTIFICATION} {IS} control-id
{ID            } {= }

{PICTURE } IS picture-string
{PIC}

{FROM } [MULTIPLE] from-item
{VALUE} [TABLE   ]

TO [MULTIPLE] to-item
```

```
{USING} [MULTIPLE] using-item
{VALUE} [TABLE   ]

OCCURS table-size TIMES

LINE [ NUMBER IS [PLUS] line-no ] [CELL  ]
                [ +  ]           [CELLS ]
                [ -  ]           [PIXEL ]
                                 [PIXELS]

{COLUMN  } [ NUMBER IS [PLUS] col-no ]  [CELL  ]
{COL     }             [ +  ]           [CELLS ]
{POSITION}             [ -  ]           [PIXEL ]
{POS     }                              [PIXELS]

CLINE NUMBER cline-num

CCOL NUMBER ccol-num

SIZE   {IS} length   [CELL  ]
       {= }          [CELLS ]
                     [PIXEL ]
                     [PIXELS]

LINES  {IS} height   [CELL  ]
       {= }          [CELLS ]
                     [PIXEL ]
                     [PIXELS]

CSIZE  {IS} clength  [CELL ]
       {= }          [CELLS]

CLINES {IS} cheight  [CELL ]
       {= }          [CELLS]

MAX-HEIGHT {IS} max-height
           {= }

MAX-WIDTH  {IS} max-width
           {= }

MIN-HEIGHT {IS} min-height
           {= }
```

```
MIN-WIDTH  {IS} min-width
           {= }

TITLE {IS} title
      {= }

KEY {IS} key-letter
    {= }

STYLE {IS} style
      {= }

{style-name} ...

FONT {IS} font-handle
     {= }

{COLOR } IS color-val
{COLOUR}

{FOREGROUND-COLOR } IS fg-color
{FOREGROUND-COLOUR}

{BACKGROUND-COLOR } IS bg-color
{BACKGROUND-COLOUR}

{BACKGROUND-HIGH     }
{BACKGROUND-LOW      }
{BACKGROUND-STANDARD }

{BELL}
{BEEP}

{HIGHLIGHT}
{HIGH     }
{BOLD     }
{LOWLIGHT }
{LOW      }
{STANDARD }

{REVERSE-VIDEO}
{REVERSED     }
{REVERSE      }

{REQUIRED   }
```

```
{EMPTY-CHECK}

LAYOUT-DATA {IS} layout-data
            {= }

ENABLED {IS} {enabled-state}
        {= }

VISIBLE {IS} {visible-state}
        {= }

HELP-ID {IS} help-id
        {= }

EVENT-LIST    {IS} ( event-value { event-value ... } )
              {= }

AX-EVENT-LIST {IS} ( ax-event-value { ax-event-value ... } )
              {= }

EXCLUDE-EVENT-LIST {IS} list-state
                   {= }

{property-name          } {IS } { property-value            }
{PROPERTY property-type } {ARE} { ( {property-value} ... )  }
                          {=  } { {MULTIPLE} property-table }
                                {TABLE   }

{BEFORE   } PROCEDURE IS { proc-1 [ {THROUGH} proc-2 ] }
{AFTER    }                        {THRU   }
{EXCEPTION}              { NULL                         }
{EVENT    }
```

Format 3

```
level-number [ screen-name ]
             [ FILLER      ]

{assembly-name}
{OBJECT assembly-name}

[ title ]

NAMESPACE { IS } "namespace"
```

```
CLASS-NAME { IS } "class-name"
```

Remaining phrases are optional.

```
HANDLE { IS } handle-1

VERSION { IS } "version"

CULTURE { IS } "culture"

STRONG-NAME { IS } "strong-name"

CONSTRUCTOR { IS } CONSTRUCTOR[n] parameters...

MODULE { IS } "module"

FILE-PATH { IS } "file-path"

SIZE  {IS} length [CELL  ]
      {= }        [CELLS ]
                  [PIXEL ]
                  [PIXELS]

LINES {IS} height [CELL  ]
      {= }        [CELLS ]
                  [PIXEL ]
                  [PIXELS]
```

Syntax Rules

1. Each screen description entry must start with a *level-number* from 01 through 49.
2. Each level 01 screen description entry must have a *screen-name* specified. *Screen-name* is a user-defined word.
3. A *screen-name* may be referenced only in those contexts where it is explicitly allowed.
4. An ACCEPT statement may reference only a *screen-name* that has a TO or USING clause specified for it, or is a group item that contains such a screen entry.
5. *From-item* and *using-item* are data items.

6. *To-item* is a literal or a data item.

7. *Table-size* is an integer literal.

8. *Fg-color*, *bg-color*, *cline-num*, *ccol-num*, *clength*, *cheight*, *color-val*, *control-id*, *layout-data*, e*nabled-state*, *visible-state*, and *help-id* are integer literals or data items.

9. *Max-height*, *max-width*, *min-height*, and *min-width* are numeric data items or numeric literals.

10. *Line-no*, *col-no*, *length*, and *height* are numeric literals or data items. In Format 1, these must be integer values.

11. *Value-lit* is a alphanumeric literal or a figurative constant.

12. *Prompt-lit* is a single-character alphanumeric literal or the figurative constant SPACE, ZERO, or QUOTE.

13. *Control-type-name* is one of the control type reserved words known by the compiler.

14. *Control-type* is a numeric literal or data item. It may not be subscripted or reference modified.

15. *Title* and *key-letter* are alphanumeric literals or data items.

16. *Font-handle* is a USAGE HANDLE or HANDLE OF FONT data item that contains a valid font handle.

17. *Style-name* is the name of a style associated with the class of control being described. If the *control-type-name* phrase is omitted, then you may not use the *style-name* phrase. The STYLE phrase may be used instead.

18. *Property-name* is the name of a property specific to the type of control being referenced. If the type of control is unknown to the compiler (as in a "DISPLAY OBJECT object-1" statement), then *property-name* may not be used. You must use the PROPERTY *property-type* option instead.

19. *Property-type* is a numeric literal or data item. It may not be subscripted or reference modified. It identifies the property to use. The numeric values that identify the various control properties can be found in the COPY library "controls.def".

20. *Property-value* is a literal or data item. Note that the parentheses in the phrase are required.

21. *Property-table* is a data item that appears in a one-dimensional table. No index should be specified.

22. In Format 1, if the PICTURE clause is used, then at least one of the FROM, TO, or USING clauses must also be used. The VALUE clause cannot be used. In Format 2, if the PICTURE clause is used, then you must specify one of the FROM, TO, USING, or VALUE phrases.

23. The MULTIPLE option and the PICTURE phrase cannot be used in the same entry.

24. The JUSTIFIED and BLANK WHEN ZERO clauses may be specified only if the PICTURE clause is also specified.

25. The COLOR clause may not be specified if either the FOREGROUND-COLOR or BACKGROUND-COLOR clause is specified.

26. You may not use either the FROM or TO phrase if you use the USING phrase.

27. In a Format 1 entry, if the VALUE phrase is used, then its meaning depends on the following data element. If it is a literal, then VALUE is synonymous with FROM. Otherwise, it is synonymous with USING.

28. If you use the MULTIPLE option of either the FROM, TO, or USING phrase, the following data element must contain an OCCURS clause or be subordinate to an OCCURS clause. The corresponding table must be one-dimensional. The data element should not be subscripted.

29. The following items may reference a table containing the appropriate type of data items, providing its entry is subordinate to an OCCURS clause: COLOR, HELP-ID, VISIBLE, ENABLED, ID, STYLE, FONT, TITLE, LINE, COL, SIZE, LINES, CCOL, CLINE, CSIZE, CLINES, KEY, and PROPERTY. See the description of the **OCCURS Clause**.

30. In Format 1, HELP-ID, VISIBLE, and ENABLED may be specified only for group items. The effect is to apply the phrase to each control contained in the group. You can override the setting for a particular

control or sub-group by specifying another HELP-ID, VISIBLE, or ENABLED phrase. These phrases have no effect on screen items that are not controls.

31. *Event-value* and *ax-event-value* are numeric literals or data items that identify an event type. List elements must be enclosed by parentheses. Elements must be separated by a space. If the list contains a single element, the parentheses can be omitted.

32. *List-state* is an integer literal or numeric data item. Valid values are "0" and "1".

33. *Assembly-name* is the name of a .NET assembly defined in a COPY file created by NETDEFGEN. This must be the DLL name of a graphical control, not an executable file. Graphical controls are generated by Visual Studio when a developer selects a "Windows Control Library" project type.

34. *Handle-1* is a USAGE HANDLE or PIC X(10) data item.

35. A value surrounded by quotation marks is an alphanumeric literal and is case-sensitive. Literal values for assembly parameters are located in the COPY file generated by NETDEFGEN. The same COPY file must be included in the SPECIAL-NAMES paragraph of your program.

36. The optional phrases may be specified in any order.

37. CELL(S) and PIXEL(S) are mutually exclusive for the same phrase.

38. In Format 2, the PLUS phrase requires PIXEL(S) to follow if the preceding LINE Number or COLUMN Number also used PIXEL(S).

39. You may mix PIXEL and conventional coordinates/sizing in the same statement, as shown here:

```
DISPLAY push-button AT 00100200 PIXELS
         LINES 50 PIXELS
         SIZE 5.
```

40. CELL and CELLS are equivalent.

41. PIXEL and PIXELS are equivalent.

42. IS and "=" are synonymous.

Note: Because the Screen Section is not part of ANSI-standard COBOL, there is substantial variation in the syntax supported by various COBOL vendors. ACUCOBOL-GT supports a superset of most other Screen Section implementations. This situation results in a large number of reserved words with the same meaning. These synonyms are detailed in each of the following rules. We recommend that you restrict yourself to one of the synonyms for each option in order to improve your program's clarity.

43. AUTO, AUTO-SKIP, and AUTOTERMINATE are equivalent.
44. NO-ECHO, NO ECHO, OFF, and SECURE are equivalent.
45. PIC is an abbreviation for PICTURE.
46. JUST is an abbreviation for JUSTIFIED.
47. BLANK and ERASE are equivalent.
48. COLUMN, COL, POSITION, and POS are equivalent.
49. BELL and BEEP are equivalent.
50. UNDERLINE and UNDERLINED are equivalent.
51. HIGHLIGHT, HIGH, and BOLD are equivalent.
52. LOWLIGHT and LOW are equivalent.
53. BLINK and BLINKING are equivalent.
54. REVERSE-VIDEO, REVERSE, and REVERSED are equivalent.
55. REQUIRED and EMPTY-CHECK are equivalent.
56. FULL and LENGTH-CHECK are equivalent.
57. MULTIPLE and TABLE are equivalent.

General Rules

1. When a *screen-name* is referenced by an ACCEPT or DISPLAY statement, that screen entry and all subordinate screen entries are acted upon at once. This allows you to accept or display many fields with one statement.

2. The word NULL in the PROCEDURE phrase indicates that there is no procedure. It has the same effect as omitting the PROCEDURE phrase altogether and is essentially commentary.

3. Screen Section entries may include the labels “GRAPHICAL” and “CHARACTER”. These markings have the effect of restricting the display of the elements nested within them. The elements contained in a GRAPHICAL Screen Section entry are displayed only when the program is run on a graphical system. The contents of a CHARACTER Screen Section entry are displayed only when the program is run on a character-based system. When the program attempts to execute a marked entry on a system of the opposite type, that entry is ignored.

   The purpose of these phrases is to better allow you to develop and support two distinct user interfaces in one program; a user interface for graphical systems, and a user interface for character-based systems. The GRAPHICAL and CHARACTER labels allow you to place and maintain all of the screen definition code in one place (the Screen Section), while also allowing you to customize the look and function of the user interface for these two classes of systems. For many developers, this approach is easier and produces more satisfying results than attempting to develop a single, *generic* user interface that works well on both types of systems.

   The following code provides an example of this approach. Suppose you want a label that describes a set of function keys to be displayed along the bottom of the screen. However, when you run the program on a graphical system you want to display push buttons instead.

   A Screen Section entry to do this might look like:

```
01  function-key-screen.
    03  line 24.
    03  character.
        05  "F1 = Exit, F2 = Lookup, F3 = Help".
```

```
03  graphical.
    05  push-button, "E&xit", self-act,
        exception-value = 1,
        column 3.
    05  push-button, "&Lookup", self-act,
        exception-value = 2,
        column + 3.
    05  push-button, "&Help", self-act,
        exception-value = 3,
        column + 3.
```

When the program executes a "DISPLAY FUNCTION-KEY-SCREEN" statement, it displays the line of text on character-based systems, or the set of push buttons on a graphical system.

Note: The use of these labels also allows you to create two screen description entries with the same name. In statements where a Screen Section name is allowed, you may now reference an ambiguous (duplicate) Screen Section name. When you do so, the name must resolve to exactly two Screen Section items, one having the CHARACTER attribute and the other having the GRAPHICAL attribute. The compiler constructs conditional code for these cases. The Screen Section item with the CHARACTER attribute is used when the program runs on a character-based system; the GRAPHICAL item is used on graphical host systems. One use for this feature is to keep a program's original screen layouts for use on character systems while creating all new screens for graphical systems. By giving the screens the same name, you can keep the existing processing logic unchanged.

Format 1

1. A Format 1 statement describes any of three types of screen description entries:

   a. If a VALUE clause is specified, the screen entry is a display literal.

   b. If a PICTURE, TO, FROM, or USING clause is specified, then the screen entry is a data field.

   c. If VALUE, PICTURE, TO, FROM, or USING is not specified, then the screen entry is a group item.

2. A LINE, COLUMN, BLANK, or BELL clause specified for a group item is acted upon immediately when that group item is accessed by an ACCEPT or DISPLAY statement. All other clauses specified for a group item are applied to each screen entry subordinate to that group item.

3. If the same clause is specified more than once for a particular screen entry, the clause specified at the lowest level within the hierarchy is the one which takes effect.

4. The *level-number*, *screen-name*, USAGE, SIGN, JUSTIFIED, and BLANK WHEN ZERO clauses follow the rules for data items appearing in Working Storage. These rules appear in **section 5.7.1, "Data Description Entry."**

5. The PICTURE, OCCURS, and VALUE clauses have meanings similar to those clauses in Working Storage, but have some additional properties when used in the Screen Section. These clauses, along with the LINE and COLUMN clauses, are detailed in separate sections below.

6. The HELP-ID, VISIBLE, and ENABLED phrases may be specified only for group items. The effect is to apply the phrase to each control contained in the group. You can override the setting for a particular control or sub-group by specifying another HELP-ID, VISIBLE, or ENABLED phrase. These phrases do not affect screen items that are not controls.

7. All other clauses are described in detail in **section 6.4.9, "Common Screen Options."**

Format 2 (SCREEN CONTROLS)

1. A Format 2 Screen Section entry defines a screen control.

2. *Control-type-name* identifies the type of the control (the exact set of controls and their types is discussed in **Chapter 5** of Book 2, *User Interface Programming*). Use the OBJECT *control-type* phrase when the type of the control is not known at compile time. *Control-type* must contain the identifying number of a control type known to the system. If it does not correspond to any control type, the Screen Section entry is ignored. The identifying number of each control type is defined in the "controls.def" file.

3. When you DISPLAY a Screen Section control, the following steps are performed:

   a. If this is the first DISPLAY of this control, the runtime builds a new control of the specified type.

   b. The various properties specified by the Screen Section control phrases are set. Unspecified properties are then assigned default values when the control is initially created. Properties are assigned in the order listed in the Screen Section, except that the VALUE property is always assigned last.

   c. The control is displayed or updated on the screen.

   d. The cursor is positioned after the control.

4. When you ACCEPT a Screen Section control, that control receives the input focus, and the runtime system processes user actions until the user terminates the ACCEPT according to the rules for the ACCEPT verb.

5. When you DISPLAY a Screen Section group item, each subsidiary Screen Section entry is displayed. This can be a mix of textual fields and graphical controls. When you ACCEPT a Screen Section group item, the cursor (or input focus) is placed according to the rules for the ACCEPT verb, and the runtime proceeds to accept data from the user for each field or control. The runtime automatically handles cursor movements between the fields and controls.

6. Screen Section controls are assigned *field numbers* in the same way as Format 1 Screen Section entries. If the control is *activatable* (the user can interact with the control), it is given a field number. Controls that cannot take user input (*e.g.*, LABEL controls) are not given field numbers. Field numbers are assigned sequentially, starting with "1", for each appropriate Format 1 or Format 2 Screen Section entry subordinate to a given 01-level group item. For Screen Section controls that omit the ID phrase, but have an implied field number, the corresponding control is given that field number as its ID. Note that the field number is not assigned until the control is created.

7. If you specify a PICTURE, the memory for that picture is allocated in the Screen Section entry. Each DISPLAY of that entry moves the data in the FROM or USING data item to itself using the standard MOVE rules. That entry is then used as the value of the control. Each

ACCEPT of that entry stores the control's value in the Screen Section entry and then moves the entry into the TO or USING data item in accordance with the standard MOVE rules. If you omit the PICTURE phrase, the control's value is retrieved directly from the FROM or USING item and stored directly in the TO or USING item. Note that specifying a PICTURE allocates additional memory. As a result, it is preferable to specify a PICTURE only in cases where you need to reformat the data (*e.g.*, by specifying a numeric-edited PICTURE).

8. The MULTIPLE phrase in the FROM, TO, or USING clause indicates that you want the control's value to be mapped to the corresponding data item on a line-by-line basis. Each line of data in the control corresponds to one element in the data table. This should be used only with controls that have the concept of multiple lines of data (*e.g.*, an ENTRY-FIELD that contains multiple lines of text). For example, the following Working-Storage and Screen Section definitions would construct a five-line entry field on the screen, and place each line of input into a separate data item in the Working-Storage table:

```
WORKING-STORAGE SECTION.
01  ENTRY-LINES OCCURS 5 TIMES  PIC X(30).

SCREEN SECTION.
01  ENTRY-FIELD, USING MULTIPLE ENTRY-LINES,
          LINES 5, SIZE 30.
```

9. If you specify a FROM or USING item, and you do not specify a *title*, the runtime will substitute the *from-item* or *using-item* for the *title* if the corresponding control type does not take a value (*i.e.*, is a LABEL, PUSH-BUTTON, or FRAME). This allows you to associate a PICTURE with a LABEL control. Because the picture formats the value of the control, and because a LABEL does not take a value, this rule allows the *picture-string* to set the value of the label's title.

10. The REQUIRED phrase is meaningful only for controls that take alphanumeric input (*e.g.*, entry fields). When specified, it forces the user to enter non-space data into the control before the ACCEPT will terminate. The user can also terminate the ACCEPT by generating an exception. See **section 6.4.9** for more information about the REQUIRED phrase.

11. The EVENT option of the PROCEDURE phrase establishes an *event procedure* for the control. Event procedures are different from the other Screen Section embedded procedures in that an event procedure becomes part of the control when it is created, and is executed directly by the control (the BEFORE, AFTER, and EXCEPTION procedures execute as part of the flow of control of the Screen Section). An event procedure can potentially execute any time after its owning control has been created, even when the defining Screen Section item is not being ACCEPTed. For more about event procedures, see **section 5.9.6**.

12. All phrases not described here or in **section 6.4.9, "Common Screen Options"** are treated in the same manner as in a Format 1 Screen Section entry.

Format 2 Screen Section example:

```
SCREEN SECTION.

01  SEARCH-SCREEN.
    03  LABEL "Search for", LINE 1, COL 5.
    03  ENTRY-BOX USING SEARCH-TEXT, COL + 1,
           SIZE 30.
    03  PUSH-BUTTON, TITLE "Ok", LINE 3, COL 10,
           DEFAULT-BUTTON, TERMINATION-CODE IS 13.
    03  PUSH-BUTTON, TITLE "Cancel", COL 25,
           CANCEL-BUTTON, EXCEPTION-CODE IS 27.
```

Format 3 (.NET ASSEMBLIES)

1. A Format 3 Screen Section entry defines a graphical .NET assembly.

2. Literal values for assembly parameters are located in the COPY file generated by the NETDEFGEN utility. The same COPY file must be included in the SPECIAL-NAMES paragraph of your program.

3. Graphical assemblies show the keyword “VISUAL” in the COPY file after the CLASS keyword. If the word “VISUAL” does not appear, use the CREATE statement to instantiate the assembly. A Format 3 Screen Section is for graphical .NET assemblies only.

4. *Assembly-name* is the name of a .NET assembly defined in the NETDEFGEN COPY file. This must be the DLL name of a graphical control, not an executable file. Graphical controls are generated by Visual Studio when a developer selects a “Windows Control Library” project type.

5. *Namespace* is a NameSpace in the assembly, as it appears in the COPY file.

6. *Class-name* is a class in the NameSpace.

7. *Handle-1* receives a handle to the assembly when it is created.

8. *Version* is the version number of the assembly.

9. *Culture* is cultural information related to the assembly.

10. *Strong-name* is the cryptographic key required to access the assembly, if any. If the assembly requires such a key, as all assemblies in the Global Assembly Cache (GAC) do, it is shown in the COPY file under the keyword STRONG.

11. All classes that result in an object have a CONSTRUCTOR, which is a unique method. If you see a CONSTRUCTOR identifier in the COPY file without a parameter list, then the field may be omitted from your COBOL program. If all listed CONSTRUCTORs have parameters, then you must choose which CONSTRUCTOR and parameters to use. *Constructor*(n) is the constructor that you want to use followed by its parameter data.

12. *Module* identifies a file where a combination of NameSpaces and Classes resides. It is used when the assembly is constructed of other assemblies.

13. *File-path* is the path of an XML file, and that XML file contains the path where the .NET assembly is located. Use FILE-PATH when the assembly that you want to access does not reside in the GAC or in the same directory as “wrun32.exe”. Assemblies that reside in the GAC will have the STRONG keyword in the NETDEFGEN COPY file.

14. LINES and SIZE default to the design control height and width.

Format 3 Screen Section example:

```
SCREEN SECTION.
01  screen-1.
    03  SOME-NETCONTROL, "@My.Assembly",
        LINE 1, COL 2,
NAMESPACE IS "My.Test.Namespace",
CLASS-NAME IS "UserControl1",
      CONSTRUCTOR IS CONSTRUCTOR2(PARM1, PARM2, PARM3,
      PARM4, PARM5, PARM6, PARM7),
EVENT PROCEDURE IS USERCONTROL-EVENTS.
```

5.9.1 PICTURE, FROM, TO, and USING Clauses

These Screen Section clauses describe the format, storage, and action of a screen data field.

General Format

```
[ {PICTURE} IS picture-string  ]
  {PIC    }

[ [ FROM out-item ] [ TO in-item ]  ]
[       USING update-item           ]
```

Syntax Rules

1. The rules for the Screen Section PICTURE clause are the same as the rules for the standard PICTURE clause detailed in **section 5.7.1.6**.

2. If the PICTURE clause is specified, then a VALUE clause may not be specified.

3. *Out-item* is a literal or an identifier referencing a data item in the File, Working-Storage, or Linkage sections. *Out-item* may be subscripted and reference modified.

4. *In-item* and *update-item* are identifiers referencing data items in the File, Working-Storage, or Linkage sections. These items may be subscripted and reference modified.

5. *In-item*, *out-item* and *update-item* must be such that a MOVE specified between them and the screen entry is legal according to the rules of the MOVE statement.

General Rules

1. Because of the difficulty in viewing the internal storage of certain classes of numeric items, you should avoid using the picture symbol "V" in a screen entry. If you use the picture symbol "S", you should also specify the SIGN SEPARATE clause. Alternately, you can use a numeric edited screen entry to display these types of data items in a sensible format.

2. A screen entry specifying the FROM phrase is an *output* field. A screen entry specifying the TO phrase is an *input* field. A screen entry specifying both the FROM and TO phrases is an *update* field.

3. A screen entry with the USING phrase is equivalent to a screen entry specifying both the FROM and TO phrases referencing the same data item.

4. When a DISPLAY verb executes, each output and update field referenced is sent to the screen. When this occurs, each *out-item* and *update-item* is first moved to the corresponding screen entry using the standard rules of the MOVE statement. These fields are then displayed on the user's screen.

5. Input fields are initialized when a DISPLAY verb executes. Numeric and numeric-edited fields have ZERO moved to them, all other fields have SPACES moved to them. These moves occur using the standard rules of the MOVE statement. The appearance of input fields on the screen when a DISPLAY verb executes is configurable by various runtime options. Note that the screen entry is initialized; the corresponding *in-item* or *update-item* is not.

6. When an ACCEPT verb executes, each input and update field referenced is input by the user. After the user is done, each screen entry is moved to the corresponding *in-item* or *update-item* using the standard rules of the MOVE statement.

7. If the PICTURE phrase is omitted, then the screen entry derives its PICTURE from the *in-item*, *out-item*, or *update-item* specified. The derived PICTURE is identical to the PICTURE for the specified item.

If both *in-item* and *out-item* are specified, then the PICTURE is derived from *out-item*. Note that only the PICTURE is derived; other clauses such as the SIGN clause are not inherited.

5.9.2 VALUE Clause

The Screen Section VALUE clause specifies a literal screen item.

General Format

```
VALUE IS value-lit
```

Syntax Rules

1. *Value-lit* is any alphanumeric literal or figurative constant.
2. If a VALUE clause is specified, a PICTURE clause may not be specified.

General Rules

1. A VALUE clause specifies a literal display field.
2. When a DISPLAY verb executes that references a screen entry with a VALUE clause, *value-lit* is sent to the user's screen.
3. If *value-lit* is a figurative constant, one occurrence of that constant is sent.

5.9.3 OCCURS Clause

The Screen Section OCCURS clause simplifies the handling of repeated data items.

General Format

```
OCCURS table-size TIMES
```

Syntax Rules

1. *Table-size* is an integer that specifies the number of occurrences of the screen entry.
2. An OCCURS clause may not be specified for a screen entry that has a level-number of 01.
3. An OCCURS clause may be subordinate to a group item with an OCCURS clause to create a two-dimensional table. An OCCURS clause may not be nested more than two deep (three dimensional or greater tables are not allowed).
4. If an OCCURS clause applies to a screen output or update field, then one OCCURS clause specifying the same number of occurrences, or no OCCURS clause at all, must apply to the source item. This OCCURS clause must not include the DEPENDING phrase.
5. If an OCCURS clause applies to a screen input or update field, then one OCCURS clause specifying the same number of occurrences must apply to the receiving item. This OCCURS clause must not include the DEPENDING phrase.
6. If a COLOR clause is specified for a screen description entry that contains or is subordinate to an OCCURS clause, that COLOR clause may reference a table of numeric data items. The number of occurrences in the color table must match the number of occurrences in the screen entry.

General Rules

1. The general rules that apply to an OCCURS clause specified for a data item in the File, Working-Storage, or Linkage sections also apply to an **OCCURS Clause** specified in the Screen Section.
2. Each screen entry affected by an OCCURS clause is repeated *table-size* times.
3. If line or column numbers are given to screen items in a table, at least one of these numbers should specify relative positioning. If absolute positioning is used, then every occurrence of the screen item will appear in the same place.

4. If the screen item is an output or an update field, and no OCCURS clauses apply to the sending item, then a DISPLAY verb causes the sending item to be moved to every occurrence of the screen item.

5. If the screen item is an output or an update field with an OCCURS clause applying to the sending item, then a DISPLAY verb causes each occurrence of the sending item to be moved to the corresponding occurrence of the screen item.

6. If the screen item is an input or an update field, then an ACCEPT verb causes each occurrence of the screen item to be moved to the corresponding occurrence of the receiving item.

7. If a COLOR clause that references a table is specified, each occurrence of the table specifies the color for the corresponding occurrence of the screen item. For example:

```
DATA DIVISION.
WORKING-STORAGE SECTION.

01  DATA-TABLE.
    03  DATA-ELEMENT OCCURS 5 TIMES PIC X(5).

01  COLOR-TABLE.
    03  COLOR-ELEMENT OCCURS 5 TIMES PIC 9(5).

SCREEN SECTION.

01  SCREEN-1.
    03  OCCURS 5 TIMES, USING DATA-ELEMENT
        COLOR COLOR-ELEMENT.
```

Examples

Table on one line

This program accepts a simple table on one line:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. OCCURS-SAMPLE-1.
DATA DIVISION.
WORKING-STORAGE SECTION.

01  TABLE-1.
    03  TABLE-ITEM OCCURS 5 TIMES     PIC X(5).
```

```
SCREEN SECTION.

01  SCREEN-1.
    03  "TABLE ITEMS:".
    03  OCCURS 5 TIMES USING TABLE-ITEM, COLUMN + 2.

PROCEDURE DIVISION.

MAIN-LOGIC.

   DISPLAY WINDOW ERASE.
   DISPLAY SCREEN-1.
   ACCEPT SCREEN-1.
   STOP RUN.
```

Two-element table

This program accepts a two-element table, each pair on its own line:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. OCCURS-SAMPLE-2.

DATA DIVISION.
WORKING-STORAGE SECTION.

01  TABLE-1.
    03  TABLE-GROUP OCCURS 5 TIMES.
        05  ITEM-1    PIC X(5).
        05  ITEM-2    PIC 9(5).

SCREEN SECTION.

01  SCREEN-1.
    03  "TEXT  NUMBER".
    03  OCCURS 5 TIMES.
        05  USING ITEM-1, LINE + 1.
        05  USING ITEM-2, COLUMN + 2.

PROCEDURE DIVISION.
MAIN-LOGIC.
   DISPLAY WINDOW ERASE.
   DISPLAY SCREEN-1.
   ACCEPT SCREEN-1.
   STOP RUN.
```

Example with output

You don't have to use a table data-item when you're doing output:

```
SCREEN SECTION.
01  LINES-SCREEN.
    03  "-" OCCURS 10 TIMES.
```

This would cause "----------" to be displayed. This is an example of rule 4, where no OCCURS clause applies to the source item.

5.9.4 LINE Clause

The LINE clause specifies the row on which the screen entry is to be placed.

General Format

```
LINE [ NUMBER IS  [PLUS] line-no ]
                  [ +  ]
                  [ -  ]
```

Syntax Rules

1. *Line-no* is a numeric literal or data item.
2. *Line-no* may not be subscripted or reference modified.

General Rules

1. The LINE clause specifies the screen row on which to place the screen entry. Line number 1 is the line number specified in the AT phrase of the ACCEPT or DISPLAY statement. If the AT phrase is not specified, then line number 1 is the first line of the current screen window.
2. The LINE clause without the PLUS option specifies an absolute line number.
3. The LINE clause with the PLUS option specifies a line number relative to the previous screen entry in the group. If the PLUS or "+" option is used, *line-no* is added to the previous line number. If the "-" option is used, *line-no* is subtracted from the previous line number.

4. If relative positioning is specified for a level 01 screen entry, the position is relative to LINE 1.
5. If *line-no* is not specified, then LINE PLUS 1 is implied.
6. If the LINE clause is omitted:
   a. If no previous screen item has been defined, LINE 1 is assumed.
   b. If a previous screen item has been defined, the line of that previous item is used.

5.9.5 COLUMN Clause

The COLUMN clause specifies which screen column to use for a screen entry.

General Format

```
{COLUMN  } [ NUMBER IS  [PLUS] col-no ]
{COL     }              [ +  ]
{POSITION}              [ -  ]
{POS     }
```

Syntax Rules

1. *Col-no* is a numeric literal or data item.
2. *Col-no* may not be subscripted or reference modified.
3. COLUMN, COL, POSITION, and POS are equivalent.

General Rules

1. The COLUMN clause specifies the screen column on which to place the screen entry. Column number 1 is the column number specified in the AT phrase of the ACCEPT or DISPLAY statement. If the AT phrase is not specified, then column number 1 is the first column of the current screen window.
2. The COLUMN clause without the PLUS option specifies an absolute column number.

3. The COLUMN clause with the PLUS option specifies a column number relative to the end of the previous screen entry in the group. If the PLUS or "+" option is used, *col-no* is added to the previous column number. If the "-" option is used, *col-no* is subtracted from the previous column number.

4. If relative positioning is specified for a level 01 screen entry, the position is relative to COLUMN ZERO (in other words, COLUMN PLUS 1 is the same as COLUMN 1).

5. If *col-no* is not specified, then COLUMN PLUS 1 is implied.

6. If the COLUMN clause is omitted:

   a. If a LINE clause is specified, COLUMN 1 is implied.

   b. If the LINE clause is omitted, COLUMN PLUS 1 is implied.

7. In ICOBOL compatibility mode, the COLUMN phrase is interpreted slightly differently. Relative positioning is based from the character just to the right of the previous data item. This causes "COLUMN PLUS 1" to place a space between the end of the last data item and the beginning of the next one. Also, if the COLUMN phrase is omitted and would normally default to COLUMN PLUS 1, then COLUMN PLUS ZERO is used instead.

5.9.6 PROCEDURE Clause

A Screen Section entry may refer to paragraphs and sections in the Procedure Division. The reference describes a procedure that the runtime will execute when a Format 2 ACCEPT statement transfers control to or from that field. The procedure temporarily suspends the operation of the ACCEPT statement. When the procedure finishes, control returns to the ACCEPT statement.

Procedures named in the Screen Section in this way are called ***embedded procedures***. You can use embedded procedures to perform immediate validation of user-supplied data.

Another type of procedure named in the Screen Section is an ***event procedure***, which is tied to a screen control. When a control generates events, it executes its event procedure as one of its first operations. When the procedure terminates, the control resumes execution. You can use event

procedures to detect and act on desired events. Event procedures are explained in detail in the General Rules section below. For a description of the specific events that can be generated in an event-driven environment, see section 4.2 of Book 2, *User Interface Programming*.

General Rule 1 describes when an AFTER or EXCEPTION procedure is executed.

You create embedded procedures or event procedures by using syntax in the Screen Section. On any Screen Section entry, you may use the following syntax.

General Format

```
{BEFORE   } PROCEDURE IS {proc-1 [{THROUGH} proc-2]}
{AFTER    }                       {THRU   }
{EXCEPTION}             {NULL                    }
{EVENT    }
```

Syntax Rules

1. *Proc-1* and *proc-2* must be names of paragraphs or sections defined in the Procedure Division of the program.
2. THROUGH and THRU are synonymous.
3. You may have one event procedure and up to three embedded procedures defined for each Screen Section entry—one BEFORE, one AFTER, and one EXCEPTION. You may not have more than one of each type in any one Screen Section entry.

General Rules

Embedded Procedures

1. An ACCEPT statement executes embedded procedures when the cursor reaches the field defined by the associated Screen Section entry. The type of an embedded procedure defines exactly when it will execute:
   a. BEFORE procedures execute when control is transferred to the associated field, before the user can enter any data.

b. AFTER procedures execute when the user attempts to leave the field normally. This can be due to typing a termination key, filling an auto-terminate field, or typing a field-movement key such as the Tab key. Keys that serve as both movement and exception keys, such as the default definition of the Up arrow key, always cause the AFTER procedure to execute.

   As pertains to this rule, the movement actions are defined as:

   Default-Next, Down, Erase-All, Erase-Next, First, Last, Left, Next, Next-Line, Numeric-Next, Previous, Previous-Line, Right, and Up.

   In the default keyboard configuration, this affects the handling of the Up and Down arrows in the first and last fields of a Screen Section item.

   To avoid having keys that are both movement keys and exception keys, you can make the affected keys movement and termination keys. In the default keyboard configuration, you would change the Up and Down arrows to be:

```
KEYBOARD   Edit=Up    Terminate=52  ku
KEYBOARD   Edit=Down  Terminate=53  kd
```

c. EXCEPTION procedures execute when the user types an exception key (assuming that exception keys are allowed) or when some other exception condition exists.

2. The word NULL in the PROCEDURE phrase indicates that no procedure exists. It has the same effect as omitting the PROCEDURE phrase and is essentially commentary.

3. When the ACCEPT statement executes an embedded procedure, it treats *proc-1* and *proc-2* in the same manner that a PERFORM statement does. That is, it begins execution at *proc-1* and continues to the end of *proc-2* (or *proc-1* if you omit *proc-2*). When the embedded procedure completes, control returns to the ACCEPT statement.

4. An embedded procedure may contain other ACCEPT statements, which, in turn, can contain embedded procedures. This is handled in the same fashion as nested PERFORMS. Embedded procedures may CALL subprograms. An embedded procedure remains active until one of the following occurs:

a. The embedded procedure finishes and returns control to its ACCEPT statement.

b. The program containing the embedded procedure does an implicit or explicit EXIT PROGRAM.

c. The run-unit containing the embedded procedure does an implicit or explicit STOP RUN.

5. Prior to executing an embedded procedure, the ACCEPT statement moves the contents of each input and update field to its corresponding data item. In addition, each control is INQUIREd for its current VALUE (unless the TC_CONTROL_SYNC_LEVEL configuration variable is set to a value that contradicts this behavior). This allows you to examine the current Screen Section data from inside an embedded procedure.

6. The ACCEPT statement updates any CURSOR variable that you have declared in Special-Names. It does this prior to executing any AFTER or EXCEPTION procedure. Note that BEFORE procedures do not update the variable. This is due to the fact that the cursor does not actually move to the field being entered until the BEFORE procedure returns control to the ACCEPT statement.

7. The ACCEPT statement updates any CRT STATUS variable that you have declared prior to executing an AFTER or EXCEPTION procedure. This allows you to determine what key the user typed to leave the field. You may also use the ACCEPT FROM ESCAPE KEY verb inside an embedded procedure for this purpose.

   a. If the user leaves the field due to the action of an editing key (such as an arrow key or a “next field” key), the CRT STATUS will be set to zero (for a numeric CRT STATUS) or to “0”, “0”, x”00” (for a group item CRT STATUS). For the format of a group item CRT STATUS, see **section 4.2.3, “Special-Names Paragraph.”**

   b. Note that some keys are both editing keys and termination keys. For example, by default, the Tab key is set up to be a “next field” key until the last field of the Screen Section, when it becomes a termination key. These keys are treated as editing keys until their termination condition applies. Therefore, their CRT STATUS value will change depending on whether the key is being treated as an editing key or a termination key.

8. After completing an embedded procedure, the ACCEPT statement re-computes all variable information contained in the Screen Section entry being accepted. This allows you to change colors and other aspects of the Screen Section entry dynamically.

    a. Note, however, that data is not automatically moved to update (USING) fields from their corresponding data items. If you want to change the contents of an update field from inside an embedded procedure (to provide a new default, for example), you must DISPLAY the changed field. The DISPLAY verb moves data to the Screen Section and shows it on the screen.

    b. In an embedded procedure, you may "protect" screen fields by changing their COLOR value (see **section 6.4.9, "Common Screen Options"**). As a result, the ACCEPT statement may terminate when the embedded procedure returns control because there are no remaining fields to enter. If this occurs, the CRT STATUS is left unchanged from the value it had when the embedded procedure returned.

9. An embedded procedure can alter the behavior of its controlling ACCEPT statement. See **section 4.2.3** for details on the SCREEN CONTROL data item.

10. You may specify an embedded procedure for a group item in the Screen Section. The effect is to apply that procedure to each elementary item contained in the group. Subsidiary items may specify their own embedded procedures, which take precedence over the group's embedded procedures. For more information, see the *User's Guide*, **section 6.5.5, "Using Screen Section Embedded Procedures."**

Event Procedures

1. The EVENT option of the PROCEDURE phrase establishes an event procedure for a control. Event procedures are different from Screen Section embedded procedures in that an event procedure becomes part of the control when it is created, while embedded procedures do not. An event procedure is executed directly by the control. Embedded procedures execute as part of the flow of control of the Screen Section.

An event procedure can potentially execute any time after its owning control is created, even when the defining Screen Section item is not being ACCEPTed.

2. By default, a control does *not* have an event procedure, which is like specifying an event procedure of NULL.

3. When a control invokes an event procedure, the EVENT-STATUS data item reflects the invoking event. When the procedure terminates, the previous contents of the EVENT-STATUS item are restored (this is important when nested events exist). If the program does not contain an EVENT-STATUS data item, then the event is processed normally. For more information about the EVENT-STATUS data item, see **section 4.2.3, "Special-Names Paragraph."**

4. Event procedures are similar to embedded procedures (for example, the AFTER procedure). However, you should note the following differences:

   a. Unlike embedded procedures, the values of data elements corresponding to a control's values are not updated when an event procedure is entered. You must use the INQUIRE verb to examine the current value of a control inside an event procedure.

   b. Unlike embedded procedures, the event procedure executes while processing the control, instead of after the control terminates. Every event that occurs in the control is passed to the event procedure. When the event procedure terminates, the event is processed by the control and the control continues normal processing as dictated by the value of the EVENT-ACTION element of EVENT-STATUS (see **section 4.2.3**). An advantage of event procedures is that they receive all events. You do not need to remember if the event should be processed in the AFTER or EXCEPTION procedure.

   c. Event procedures can be executed any time the control receives events. The Screen Section embedded procedures receive control only during an ACCEPT of the corresponding Screen Section. This behavior can be important with tool bar controls, which normally are not ACCEPTed.

d. Events classified as *messages* (whose symbolic names start with "MSG-") are sent only to event procedures. They cannot be detected by the Screen Section embedded procedures. See **Chapter 6** of Book 2, *User Interface Programming*, for detailed information about messages.

e. You can change a control's event procedure via the MODIFY verb. The Screen Section's embedded procedures are fixed.

5. Event procedures are unique in that they can receive control when the owning COBOL program is otherwise inactive. The following rules cover the activation of an event procedure:

   a. If the current program contains the event procedure, then it is executed as if it were the subject of the PERFORM statement.

   b. If the program containing the event procedure is active, but is not the current program, control flows to the containing program at the point of the event procedure. This situation is not considered a recursive call, because the transfer of control is not via CALL. When the event procedure terminates, control returns to the original program.

   c. If the program containing the event procedure is not active, but is resident in memory, it is made active with the following conditions:

      - It has no USING items passed to it, so program parameters are not defined and should not be used.
      - While active, the program is not affected by CANCEL (just like all active programs).
      - When the event procedure terminates, the program is made inactive again.
      - On entry to the event procedure, all other states of the program (e.g., the values of variables) are unchanged from the time when the program was last active.
      - The program retains any changes made to its state (variables and file state) when the procedure exits.

d. If the program containing the event procedure is not memory resident, then the event procedure is removed from the control.

e. For recursive programs, the copy of the program at the depth that registered the procedure is the one that counts for determining which program contains the event procedure.

f. While an event procedure is active (due to an event), an EXIT PROGRAM statement is ignored in the event procedure's controlling program.

6. When programming an event procedure, you should avoid using the ACCEPT statement to allow input from controls. Most graphical host systems are not designed to expect input events while in the process of handling another input event. This situation stresses the host system and can lead to situations where the results are unpredictable.

   If you want to place an ACCEPT statement inside an event procedure, one way to avoid this situation is to turn the event to a terminating event and then perform the ACCEPT in the corresponding exception procedure. Exceptions occur after event handling is complete, so this avoids the "nested input events" syndrome. To turn an event into a terminating event, see the description of EVENT-STATUS, in **section 4.2.3, "Special-Names Paragraph."** Another effective technique is to run the procedure that performs the ACCEPT in a separate thread. This allows the original event to complete and also avoids the situation. Typically you can do this by placing the desired event-handling code in a separate paragraph and then using PERFORM THREAD to run that paragraph from the control's event handler.

6 Procedure Division

Key Topics

6.1 Organization

The Procedure Division holds the COBOL statements that the program executes. This chapter describes the general rules covering COBOL statements and covers each statement type in detail.

The Procedure Division is organized as a series of paragraphs made up of sentences. These sentences describe the desired behavior of the program. Paragraphs may optionally be organized into sections. Most paragraphs in the Procedure Division contain statements that are executed in the normal course of the program. Paragraphs may also be placed in *Declaratives*. Declarative paragraphs execute only in response to some external condition, such as failure of a file input/output statement.

6.1.1 Statements and Sentences

A COBOL statement is always introduced by a reserved word called a *verb*. A verb and its operands describe some action to be taken when the program runs. A *sentence* is one or more statements that are terminated by a period.

There are four types of statements:

1. *Compiler-directing* statements specify an action to be taken by the compiler. Only the COPY, REPLACE, and USE statements fit this classification.

2. *Imperative* statements specify an unconditional action to be taken by the object program. Whenever an imperative statement is allowed, it may consist of a sequence of consecutive imperative statements.

3. *Conditional* statements specify an action to be taken by the object program that is dependent on the truth value of some condition.

4. *Delimited-scope* statements specify their explicit scope delimiter. A delimited-scope statement contains elements of a conditional nature. Because of the scope-delimiter, however, these statements may be used anywhere an imperative statement may be.

An *imperative* sentence is one that contains only imperative and delimited-scope statements. A *conditional* sentence consists of a single conditional statement optionally preceded by a sequence of imperative statements.

Several verbs can be either imperative, conditional, or delimited-scope. For example, a simple READ statement is imperative. If the AT END clause is included, it becomes conditional. On the other hand, if the END-READ phrase is also included, then it is a delimited-scope statement.

6.1.1.1 Scope of statements

The scope of a statement designates when a statement ends. A period terminates all currently active statements. In a delimited-scope statement, the scope delimiter ends the statement. An imperative statement ends at the beginning of the next statement.

Conditional and delimited-scope statements can contain other statements. These statements can be terminated implicitly by elements of the containing statement. In the following example, the ELSE clause terminates the ADD and DISPLAY statements. The period terminates the MOVE and IF statements.

```
IF VAR-1 = VAR-2
    ADD VAR-X TO VAR-Y
        ON SIZE ERROR DISPLAY "OVERFLOW"
ELSE
    MOVE VAR-2 TO VAR-1.
```

Clauses or scope-delimiters that terminate a statement always terminate the statement that begins with the closest preceding unpaired verb of the appropriate type. In the following example, the first ELSE terminates the DISPLAY statement, the second ELSE terminates the MOVE statement and the second IF statement, and the period terminates the ADD statement and the first IF statement.

```
IF VAR-1 = VAR-2
    IF VAR-3 = VAR-4
        DISPLAY "CONDITION 1"
    ELSE
        MOVE "CONDITION 2" TO VAR-5
ELSE
    ADD 1 TO VAR-3.
```

A delimited-scope statement is one that contains a scope-delimiter. A scope-delimiter is always a reserved word that begins with "END-" and finishes with a verb name. For example, END-ADD and END-START are scope delimiters for the ADD and START statements. The presence of a scope-delimiter converts a conditional statement into a delimited-scope statement. A delimited-scope statement may be used anywhere an imperative statement can appear.

Note: Only imperative statements may be nested within another statement, unless the other statement is an IF statement.

The following is *illegal*:

```
ADD VAR-1 TO VAR-2 ON SIZE ERROR
       IF VAR-1 < ZERO
          DISPLAY "VAR-1 IS NEGATIVE"
       ELSE
          DISPLAY "VAR-1 TOO LARGE".
```

Since the IF statement is conditional, it may not be nested within the ADD statement. The correct version of this construct is:

```
ADD VAR-1 TO VAR-2 ON SIZE ERROR
   IF VAR-1 < ZERO
      DISPLAY "VAR-1 IS NEGATIVE"
   ELSE
      DISPLAY "VAR-1 TOO LARGE"
   END-IF.
```

The presence of the END-IF converts the conditional IF statement into a delimited-scope statement which may be used in place of an imperative statement. Thus it may appear nested within the ADD statement.

6.1.2 Flow of Control

Execution of a program begins at the first paragraph encountered outside of Declaratives. Statements are executed in order except when an explicit or implicit transfer of control occurs. The program halts when there is no next statement available to execute.

The following methods of implicit control are used by COBOL.

1. If a paragraph is being executed under control of another COBOL statement (such as a PERFORM) and the paragraph is the last in the range of the controlling statement, then an implied transfer of control occurs from the last statement in the paragraph to the control mechanism of the controlling statement.

2. When any COBOL statement is executed which results in the execution of a Declarative section, an implicit transfer of control to that Declarative section occurs. After the section completes, another implicit transfer of control occurs to the statement immediately after the statement that caused the Declarative to execute.

An explicit transfer of control occurs in response to certain COBOL statements such as GO TO and PERFORM. The rules for transfer are described under the appropriate verbs. All conditional and delimited-scope statements also perform explicit transfer of control based on some condition.

6.2 Arithmetic Expressions

Arithmetic expressions are used in COBOL to represent a fixed or computed numeric value. An arithmetic expression can be one of the following:

1. a numeric elementary data item
2. a numeric literal
3. the address of a data item (ADDRESS OF phrase)
4. two or more of the above, separated by arithmetic operators
5. two or more arithmetic expressions, separated by arithmetic operators
6. an arithmetic expression enclosed in parentheses
7. an arithmetic expression preceded by a unary operator (a sign)

Arithmetic expressions can use five binary and two unary arithmetic operators. A space must precede and follow each operator. The operators are:

Binary Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

Unary Operator	Meaning
+	Multiplication by +1
-	Multiplication by -1

Note: Spaces are often required before and after minus signs to prevent ambiguity.

6.2.1 Evaluation of Arithmetic Expressions

Parentheses may be used in expressions to specify the order of evaluation. Expressions within parentheses are evaluated first. When parentheses are nested, the innermost set of parentheses is evaluated first, and then successively more inclusive parentheses are evaluated.

When an expression contains no parentheses, the expression evaluates the arithmetic operators in the following hierarchical order:

1. unary plus and minus
2. exponentiation
3. multiplication and division
4. addition and subtraction

When the sequence of execution is not specified by parentheses and two or more operators exist at the same hierarchical level, the order of evaluation is from left to right.

An arithmetic expression can begin with only a left parenthesis, a plus sign, a minus sign, an identifier, or a literal. It can end only with a right parenthesis, an identifier, or a literal. Each left parenthesis in an expression must have a matching right parenthesis, and each right parenthesis must have a matching left parenthesis. If the first operator is unary, it must be preceded by a left parenthesis if the expression immediately follows an identifier or another arithmetic expression.

Operands in an arithmetic expression may be any format or USAGE.

6.2.2 ADDRESS OF Phrase in Expressions

In an arithmetic expression, you may use the address of a data item anywhere you can normally place a numeric data item. The address is treated as an unsigned integer with sufficient range to express any address in the host machine's address space.

To create an address, you use the following syntax:

```
ADDRESS OF data-item-1
```

where *data-item-1* is a data item.

For example, the following statement produces a pointer that points one character position past the beginning of ITEM-1:

```
COMPUTE PTR-1 = ADDRESS OF ITEM-1 + 1
```

If a data item has not been given an address by the program, then the data item's address acts as if its uppermost parent item (or itself if it has no parent) has an address of zero. This makes it possible to test for a NULL address without generating a runtime exception. For example:

```
SET POINTER-1 TO ADDRESS OF LINK-ITEM-1
IF POINTER-1 = NULL
   SET ADDRESS OF LINK-ITEM-1 TO WS-ITEM-1
```

This rule only applies to ADDRESS OF calculations.

6.3 Conditional Expressions

Conditional expressions specify a condition the program must evaluate to determine the program's behavior. Conditional expressions have a value of "true" or "false". Conditional expressions can either be simple or complex. The simple conditional expressions are the relation, class, sign, and condition-name conditions. The complex conditions are formed with the logical operators AND, OR, and NOT.

6.3.1 Relation Conditions

A relation condition specifies a comparison of two operands, each of which may be a data item or a literal. A relation condition has the value "true" if the relation exists between the operands. Comparison of two numeric operands is permitted regardless of the formats specified by their respective USAGE clauses. For all other comparisons, the operands must have the same USAGE.

The format of a relation condition is:

```
value1 IS [NOT]  { GREATER THAN               }  value2
                 { >                          }
                 { LESS THAN                  }
                 { <                          }
                 { EQUAL TO                   }
                 { =                          }
                 { <>                         }
                 { GREATER THAN OR EQUAL TO   }
                 { >=                         }
                 { LESS THAN OR EQUAL TO      }
                 { <=                         }
```

In the preceding format, *value1* and *value2* may be either an arithmetic expression, a data name, or a literal (including NULL or NULLS). The relational operators have the following meanings:

Meaning	**Operators**
Greater than	IS GREATER THAN IS > IS NOT <= IS NOT LESS THAN OR EQUAL TO
Equal to	IS EQUAL TO IS =
Less than	IS LESS THAN IS < IS NOT >= IS NOT GREATER THAN OR EQUAL TO
Greater than or equal	IS NOT LESS THAN IS NOT < IS >= IS GREATER THAN OR EQUAL TO
Not equal to	IS NOT EQUAL TO IS NOT = IS <>
Less than or equal	IS NOT GREATER THAN IS NOT > IS <= IS LESS THAN OR EQUAL TO

6.3.1.1 Comparison of numeric operands

For operands whose class is numeric, a comparison is made with respect to the algebraic value of the operands. The size of the operands (in terms of number of digits) is not significant. Zero is a unique value regardless of sign. Comparison is allowed regardless of the USAGE of the operands. Unsigned operands are considered positive or zero.

Index data items are compared as if they were numeric operands.

6.3.1.2 Comparison of nonnumeric operands

For nonnumeric operands, or one numeric and one nonnumeric operand, a comparison is made with respect to the program's collating sequence of characters.

If one of the operands is numeric, it must be an integer data item or an integer literal. The numeric operand is treated as though it were moved to an alphanumeric data item of the same size as the numeric data item. This alphanumeric item is then used in the comparison. A non-integer numeric operand may not be compared to a nonnumeric operand.

When the comparison is performed, the shorter operand (if any) is treated as though it were extended on the right with spaces to make the operands of equal size. Characters are then compared in corresponding positions starting from the left end and continuing until either a pair of unequal characters is encountered or the right end of the operand is reached. If all the characters are the same, then the operands are considered equal. Otherwise, the operand with the smaller character in the first unequal pair is considered less than the other operand.

6.3.2 Class Condition

The class condition tests whether or not an operand contains a particular type of data. The general format of the class condition is:

```
variable IS [NOT] {NUMERIC         }
                  {ALPHABETIC      }
                  {ALPHABETIC-UPPER}
                  {ALPHABETIC-LOWER}
                  {class-name      }
```

Variable must name a data item with USAGE DISPLAY. If *class-name* is used, it must be a name defined in a CLASS clause in the SPECIAL-NAMES paragraph.

The NUMERIC test cannot be used with an elementary item whose class is alphabetic or a group item composed of elementary items which contain operational signs. If the item being tested is *not* described as having an operational sign, the item is considered numeric only if the content consists

of the digits “0” through “9”. If the item *is* described as having an operational sign, the item is considered numeric only if the content consists of the digits “0” through “9” and a valid operational sign.

The ALPHABETIC test cannot be used with an item whose class is numeric. The test is true if the content of the data item consists entirely of the characters “A” through “Z”, “a” through “z”, and space. The ALPHABETIC-LOWER and ALPHABETIC-UPPER tests are similar to the ALPHABETIC test except that only lower-case or upper-case characters are allowed respectively.

If *class-name* is used, then *variable* may not be numeric. The test is true if *variable* consists solely of characters named in the CLASS clause that defined *class-name*. See **section 4.2.3, “Special-Names Paragraph,”** for a description of the CLASS clause.

If the NOT option is specified, then the test is true if and only if the same condition without the NOT option is false.

6.3.3 Sign Condition

The sign condition tests whether an arithmetic expression is positive, negative, or zero. The format of a sign condition is:

```
arithmetic-expression IS [NOT] {POSITIVE}
                               {NEGATIVE}
                               {ZERO    }
```

An arithmetic expression is POSITIVE if it is greater than zero, NEGATIVE if it is less than zero. If the NOT option is specified, the truth value of the test is reversed.

6.3.4 Condition-Name Condition

A condition-name condition tests whether a condition-variable has one of the values associated with the condition-name. The format of the condition-name condition is simply the condition-name.

If the condition-name is associated with a range or ranges of values, then the condition-variable is tested to determine whether or not its value falls in this range, including the end values. The rules of comparison are the same as those for a relation condition.

6.3.5 Switch-Status Condition

A switch-status condition tests the value of one of the external program switches. Program switches can be set at the command line when the program is run, or may be set with the SET verb. By default, each switch is initially set to “off”.

The STATUS clause of the SPECIAL-NAMES paragraph associates a switch-status name with either the “on” or “off” setting of a particular switch. When a switch is set, the corresponding ON STATUS name is “true” and the OFF STATUS name is “false”. When the switch is reset, the reverse is true. In a conditional expression, you can test a switch’s on/off status by simply specifying the desired switch-status name.

An example of code involving switches follows:

```
identification division.
program-id. testit.
environment division.
special-names.
    switch 1 is switch-1 on status is first-on off status is
first-off
    switch 2 is switch-2 on status is second-on off status is
second-off
    switch 3 is switch-3 on status is third-on off status is
third-off
data division.
procedure division.
main.
    display window erase.
    set switch 3 to off.

    if first-on
        display "First Switch On. . ." line 5 col 4
    else
        display "First Switch Off. . ." line 5 col 4.
```

```
if second-on
    display "Second Switch On. . ." line 7 col 4
else
    display "Second Switch Off. . ." line 7 col 4.

if third-on
    display "Third Switch On. . ." line 9 col 4
else
    display "Third Switch Off. . ." line 9 col 4.

accept omitted.
stop run.
```

6.3.6 Complex Conditions

A complex condition is formed by combining conditions (either simple or complex) with logical connectors (AND and OR) or by negating these conditions with logical negation (NOT). The truth value of a complex condition depends on the interaction of the logical operators and their component conditions.

The logical operators and their meanings are:

Operator	**Meaning**
AND	true when both components true
OR	true when either component true
NOT	true when condition false

A condition is negated by the logical operator NOT, which reverses the truth value of the condition to which it is applied. In other words, a negated condition is true when its component condition is false, and is false when its component condition is true. The format of a negated condition is:

```
NOT condition
```

6.3.6.1 Combined conditions

A combined condition results from connecting conditions with one of the logical operators AND or OR. The general format is:

```
condition { {AND} condition } ...
            {OR }
```

In the general format, *condition* may be any of the following:

1. a simple condition
2. a negated simple condition
3. a combined condition
4. a negated combined condition; that is, a "NOT" followed by a combined condition enclosed in parentheses

Parentheses may be used in a complex condition to alter the rules of evaluation. Parentheses must be made in matched pairs and must be placed in such a way so that the enclosed symbols constitute a well-defined condition.

The truth value of a combined condition using AND is true only when both component conditions are true. The truth value of a combined condition using OR is true when either or both of the component conditions are true.

6.3.7 Order of Evaluation

A condition is evaluated according to the following hierarchy:

1. Components contained in parentheses are evaluated first. For those components in parentheses, the most deeply nested are evaluated first, followed by those that are less deeply nested. Evaluation always progresses from the most to the least deeply nested components.
2. NOT conditions are evaluated next. Thus NOT A OR B is equivalent to (NOT A) OR B.
3. AND conditions are evaluated next.
4. OR conditions are evaluated next.
5. Conditions are evaluated from left to right.

Here are some examples of equivalent conditions:

```
NOT A AND B OR C AND D      ((NOT A) AND B) OR (C AND D)
NOT A AND B AND C           ((NOT A) AND B) AND C
A AND B OR NOT (C OR D)     (A AND B) OR (NOT (C OR D))
NOT ( NOT A OR B OR C )     NOT ( ((NOT A) OR B) OR C )
```

Evaluation of a condition halts as soon as its truth value is determined. For example, in the condition A AND B, the condition 'B' would not be evaluated if 'A' were false.

6.3.8 Abbreviated Combined Relation Conditions

When simple or negated simple conditions are combined in a consecutive sequence, the relation conditions may be abbreviated. You can do this by either:

1. omitting the subject of the relation condition (the left-hand component condition).
2. omitting both the subject and the relational operator.

The format for an abbreviated combined relation condition is:

```
condition { {AND} [NOT] [relation] object } ...
          {OR }
```

Within a sequence of relation conditions, both of the above forms of abbreviation may be used. When you use such abbreviations, it is as if the last preceding stated subject were inserted in place of the omitted subject, and the last stated relational operator were inserted in place of the omitted relational operator. The insertion of an omitted subject or relational operator terminates once a complete simple condition is encountered within a complex condition. Except for the source of the initial relation condition, no parentheses may appear in the sequence of abbreviated conditions.

If the word NOT is used in an abbreviated combined relation condition, it has the following meaning:

1. If the word immediately following NOT is GREATER, ">", LESS, "<", EQUAL, "<=", ">=", or "=", then the NOT participates as part of the relational operator.

2. Otherwise, the NOT is interpreted as a logical operator and the implied insertion of the subject or relational operator results in a negated relation condition.

The following are examples of abbreviated combined relation conditions and their expanded equivalents:

```
A > B AND NOT < C       (A > B) AND (A NOT < C)
(A > B AND NOT < C)     (A > B) AND (A NOT < C)
A NOT = B OR C          (A NOT = B) OR (A NOT = C)
NOT A = B OR C          (NOT (A = B)) OR (A = C)
NOT (A > B OR < C)       NOT ((A > B) OR (A < C))
```

–newARC compiler option

With "--newARC" compiler option,the syntax rules for Abbreviated Combined Relation conditions become relaxed to allow for parenthesis to appear immediately after the relation.

For example:

```
IF A = (B OR C OR D)
```

would now be accepted by the compiler.

6.4 Common Statement Rules

The following sections cover rules that apply to several verb types.

6.4.1 Arithmetic Operations

The arithmetic statements are ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT. They have several common features.

1. The data descriptions of the operands need not be the same. Any necessary conversion and decimal point alignment is supplied throughout the calculation.

2. The maximum size of each operand is 18 digits. This increases to 31 if 31-digit support (-Dd31) is in effect. If the composite of operands contains more than 18 digits, the results will be undefined. The composite of operands is a hypothetical data item resulting from the superposition of the operands aligned on their decimal point.

3. A calculation may result in the compiler's constructing a temporary data item to hold an intermediate result. The temporary item holds the 20 most significant digits of the intermediate result. (If 31-digit support (-Dd31) is in effect, the temporary data item holds 33 digits.) The truncated low-order digits (if any) are treated as zeros.

4. When the final value of the operation is stored in the receiving field(s), it is transferred to maintain its arithmetic value. If the receiving field is too small to hold the result, leading digits before the decimal point and trailing digits after the decimal point are removed as needed (but see the ON SIZE ERROR phrase for an exception). If the receiving field is unsigned and the value is negative, the results are undefined.

5. An attempt to use non-numeric data in a field defined as numeric results in an *intermediate* runtime error. Intermediate errors call installed error procedures. See Book 4, Appendix I, "Library Routines," for detailed discussion of the runtime **Error and Exit Procedures**.

Note: In division operations, the remainder is calculated before the quotient is moved to the destination item(s). The remainder will almost always be 0 if the dividend or divisor is floating-point. This is because all of the arithmetic is performed using floating-point variables. The remainder will be non-zero only if precision is lost during the calculation.

6.4.2 Multiple Receiving Fields

The arithmetic statements may have multiple resultant fields specified for them. These statements are evaluated as if they were written as a series of statements involving a temporary data item.

The following example illustrates the process.

The statement

```
ADD A, B, C TO C, D(C), E
```

is equivalent to

```
ADD A, B, C GIVING TEMP
ADD TEMP TO C
ADD TEMP TO D(C)
ADD TEMP TO E
```

6.4.3 ROUNDED Option

The arithmetic statements allow for the ROUNDED phrase to be optionally specified. The results of an arithmetic statement depend on whether this phrase is present or not. When this phrase is specified, the operation adds 1 to the absolute value of the low-order digit of the resultant data item, if the absolute value of the next least significant digit of the intermediate result is greater than or equal to 5. If the ROUNDED phrase is not specified, all excess digits in the intermediate result are truncated when it is moved to the resultant data item. (See **section 6.4.1, "Arithmetic Operations,"** and see the **DIVIDE Statement** in **section 6.6, "Procedure Division Statements,"** for relevant information.)

6.4.4 SIZE ERROR Option

The SIZE ERROR clause allows the programmer to specify actions to be taken when a size error occurs in an arithmetic statement. A size error condition exists under the following circumstances:

1. The absolute value of an operation's result exceeds the largest value the resultant data item can contain.
2. A division by zero is performed.
3. The value zero is raised to a power of zero.
4. A negative number is raised to a non-integer power.

The size error condition exists only for those resultant data items to which it applies. If the ROUNDED phrase is specified, it is applied before the size error condition is checked.

If a size error occurs, the results depend on whether or not the SIZE ERROR (or NOT SIZE ERROR) clause is specified. If the SIZE ERROR clause is present, the values of the data items on which a size error occurs remain unchanged. If there are multiple resultant data items in the statement, the ones on which size errors occur will be unchanged while the ones in which no size error occurs will be updated to the computed values. If a size error occurs and no SIZE ERROR clause is specified, the results are undefined. In cases where the size error is a divide by zero condition, by default the results are still undefined, however, the A_CECKDIV configuration variable can be used to specify either of two alternate runtime behaviors. For more information, see **A_CHECKDIV** in Book 4, Appendix H.

If the SIZE ERROR clause is present and a size error occurs, the corresponding statement is executed. If the NOT SIZE ERROR clause is present, its statement is executed if no size errors occur during the computation.

If the CORRESPONDING option is used in an ADD or SUBTRACT statement, any individual operation can cause a size error to occur. The SIZE ERROR clause is not executed, however, until all individual operations are completed.

6.4.5 CORRESPONDING Option

The CORRESPONDING option of the MOVE, ADD, and SUBTRACT verbs allows the programmer to specify group items as operands in order to use their corresponding subordinate data items. All identifiers in a CORRESPONDING phrase must refer to group items.

The operation specified by the statement acts on each pair of corresponding items in the specified group items. If two or more source data items have the same name, or two or more destination items have the same name, the operation may be performed more than once on some items.

Data items are considered to correspond if they match the following rules. In these rules, *ident-1* and *ident-2* refer to the two group items specified in the CORRESPONDING phrase.

1. The data items in *ident-1* and *ident-2* must have the same data-name (which may not be FILLER).
2. They must have the same qualifiers up to but not including *ident-1* and *ident-2*.
3. At least one of the data items must be elementary, and the resulting operation must be legal under the rules of the statement being executed. A violation to this rule causes a compile-time error; it does not exclude the item from the operation.
4. In ADD or SUBTRACT statements, both of the data items must refer to an elementary numeric data item.
5. Data items subordinate to a REDEFINES, RENAMES, OCCURS, or USAGE IS INDEX clause are ignored.

6.4.6 Unpredictable Results

When a sending and a receiving item in any statement share a part or all of their storage areas, the result of the statement is undefined.

6.4.7 I/O Status

After the execution of any file I/O verb (CLOSE, DELETE, OPEN, READ, REWRITE, START, UNLOCK, or WRITE), information about the I/O is available in the form of a status number. This number is placed in the FILE STATUS variable for the appropriate file if one has been declared.

Status-variable must be the name of an alphanumeric (or USAGE DISPLAY numeric) Working-Storage or Linkage data item with a size of 2 characters. (The compiler also allows *status-variable* to be a numeric item; however, in this case it emits a Warning.) The first digit indicates a general classification of the status, while the second digit provides detail information. Any file status whose first digit is “0” is considered a successful I/O. Any non-zero initial digit indicates failure.

Appendix E, Book 4, covers the exact values returned for each I/O condition.

6.4.8 AT END and INVALID KEY Phrases

Several of the I/O statements can take an optional AT END phrase and NOT AT END phrase. Both of these phrases are followed by a statement that is conditionally executed, depending on the result of the I/O statement. The discussion of each I/O statement that can take the AT END phrase specifies the conditions that cause the "at-end" condition to be in effect. If the at-end condition is in effect, one of the following happens:

1. If the AT END phrase is present, the statement it contains is executed; otherwise
2. If an appropriate Declarative section exists, it is executed; otherwise
3. The program prints a message and halts. Note, however, that the runtime can be configured to ignore the error and keep running. See the configuration variables **ERRORS_OK** and **EOF_ABORTS** in Appendix H, Book 4. See also the additional information on ERRORS_OK in **Section 2.8.5** of Book 1, *ACUCOBOL-GT User's Guide*.

If the NOT AT END phrase is present, it is executed if the I/O statement is successful. Note that it is possible to execute neither the AT END nor NOT AT END phrases if an I/O error occurs that does not set the at-end condition. In this case, the AT END phrase is not executed (because the at-end condition is not in effect) and the NOT AT END phrase does not execute (because the I/O statement is not successful).

All of the preceding comments about the AT END and NOT AT END phrases also apply to the INVALID KEY and NOT INVALID KEY phrases (except that the invalid-key condition is tested instead of the at-end condition).

6.4.9 Common Screen Options

The ACCEPT, DISPLAY, and MODIFY verbs, along with screen entries defined in the Screen Section, have several options in common. The syntax details for these options are presented in the individual sections where they are allowed. See:

- **section 5.9, "Screen Description Entry"**
- **section 6.6** for the ACCEPT, DISPLAY, and MODIFY statements

The following subheadings describe the effects of these options.

Note: Options that are specific to a single statement or have specialized meanings are described under those statements.

AUTO Phrase

```
{AUTO          }
{AUTO-SKIP     }
{AUTOTERMINATE}
```

1. The AUTO phrase causes a field entry to automatically terminate as soon as the field is filled with data. If the field is the last field in a group of screen items (or the only field), then the ACCEPT statement will terminate when this occurs.
2. If the AUTO phrase is not specified, then the field entry must be terminated by the user's typing a valid termination key (such as the enter key).
3. When you are using RM/COBOL compatibility mode, AUTO is implied by default for any Format 1 ACCEPT statement. This may be overridden with the TAB phrase (see below).

BACKGROUND-HIGH, BACKGROUND-LOW, and BACKGROUND-STANDARD Phrases

```
{BACKGROUND-HIGH     }
{BACKGROUND-LOW      }
```

```
{BACKGROUND-STANDARD}
```

1. Specifying BACKGROUND-HIGH causes the background color to be shown in high-intensity. BACKGROUND-LOW causes it to be shown in low intensity.

2. BACKGROUND-STANDARD causes fields and controls to use the current subwindow's background intensity. When a window is created, BACKGROUND-STANDARD causes the window to use the default background intensity for the host system. See the **BACKGROUND_INTENSITY** runtime configuration variable in Appendix H for more details.

3. If no background intensity is specified, the current subwindow's intensity is used.

Note: Most character-based systems cannot control the background intensity.

BELL Phrase

```
WITH [NO] {BELL}
         {BEEP}
```

The BELL phrase causes the terminal's bell to ring prior to performing the I/O. The NO BELL phrase inhibits the bell from ringing. The default is NO BELL except that in RM/COBOL compatibility mode, a Format 1 ACCEPT statement defaults to BELL. See also the information on the **BELL** configuration variable in Appendix H, Book 4.

BLINK Phrase

```
WITH  {BLINKING}
      {BLINK   }
```

Specifying the BLINK phrase causes the field to have the "blink" video attribute set for it. If the terminal hardware does not support blinking, this phrase is ignored. Note that many implementations of ACUCOBOL-GT will set the foreground and background colors for a field with the BLINK phrase to white-on-black. Note also that BLINK cannot be combined with UNDERLINED. Microsoft Windows does not support blinking.

CCOL, CLINE, CLINES, and CSIZE Phrases

```
AT CLINE NUMBER cline-num

AT CCOL NUMBER ccol-num

CSIZE   {IS} clength  [CELL ]
        {= }          [CELLS]

CLINES  {IS} cheight  [CELL ]
        {= }          [CELLS]
```

1. The CCOL, CLINE, CLINES, and CSIZE phrases provide an alternate method for specifying the placement (row and column) or size (height and width), or both, of a control for display on a non-graphical system. Together these phrases are called the *character coordinate* phrases. In any context where the COL, LINE, LINES, and SIZE phrases can be specified for a control, you can also specify the corresponding character coordinate phrase. These phrases make it easier to use controls on both graphical and non-graphical systems (see also section 3.5 of Book 2, *ACUCOBOL-GT User Interface Programming*).

2. You specify the character coordinate phrases in exactly the same fashion as their regular counterparts (i.e., CLINE is similar to LINE, CCOL to COL, etc.). All of the syntax supported by one phrase works in the corresponding phrase. For example, a Screen Section control item that has a twin set of column offsets is:

   ```
   entry-field, col + 2, ccol + 1, ...
   ```

   When the application is run on a graphical host system, the character coordinate phrases have no effect (exception: in some contexts the values are evaluated, so any relevant table indexes should be set to legal values to prevent access violations). When the application is run on a character-based host, the character coordinate phrases substitute for their counterparts. For example, if you specify both LINE and CLINE for a control, the CLINE specification will be used as the line number on a character-based system. Omitting a character coordinate phrase causes the regular counterpart to be used instead.

3. If you specify the CELLS option in either the SIZE or CSIZE phrase, then you must use the CELLS option in both phrases. The same rule applies to the use of the CELLS option with the LINES and CLINES phrases.

COLOR Phrase

```
{COLOR } IS color-val
{COLOUR}
```

1. The COLOR phrase provides an alternate method for setting video attributes. It also allows the specification of colors for screen fields and controls.

2. *color-val* can be set to different numeric values to express various combinations of colors and video attributes. You may make combinations by adding the appropriate values together. The following color values are accepted:

Color	**Foreground**	**Background**
Black	1	32
Blue	2	64
Green	3	96
Cyan	4	128
Red	5	160
Magenta	6	192
Brown	7	224
White	8	256

You may specify other video attributes by adding the following values:

Reverse video	1024
Low intensity	2048
High intensity	4096
Underline	8192
Blink	16384
Protected	32768
Background low-intensity	65536
Background high-intensity	131072

You may also specify high intensity by adding “8” to the foreground color value.

3. Only one foreground color and one background color may be specified. High intensity and low intensity may not both be specified. If neither is specified, the default intensity is used. If either the foreground or background color value is missing, then the corresponding default for the current subwindow is used. (Exception: some control types follow different rules for selecting the default color. These rules are described in Chapter 5, Book 2, *ACUCOBOL-GT User Interface Programming* in the sections that describe each control type).

   When a control is assigned a non-zero foreground or background color, that component will no longer revert to the default when assigned the value “0” (zero). Instead, assigning “0” simply reassigns the current color. The runtime uses this behavior to preserve one component (foreground or background) when changing the other.

4. Other video attribute clauses (such as the REVERSED clause) may be specified along with the COLOR clause. The effects of all such clauses are used together. If any conflict exists, the COLOR clause takes precedence.

5. Reverse video exchanges the foreground and background colors.

6. If a terminal does not support color, then all non-black colors are mapped to white at runtime. If this results in a white-on-white combination, then the background color is set to black. However, if you explicitly specify identical foreground and background colors, then the characters will be invisible.

7. The “Protected” attribute applies to only Format 1 Screen Section entries. Any input or update field that has the “Protected” attribute set for it is ignored during an ACCEPT statement. This enables you to selectively allow entry of certain data fields. The “Protected” attribute does not affect the behavior of the DISPLAY statement.

8. The “Protected” attribute cannot be applied with the MODIFY statement and the COLOR phrase. For example, the following will *not* set the field to “Protected”:

```
01   my-field, ENTRY-FIELD, LINE 1, COL 1.

MODIFY my-field, COLOR=32768
```

Instead, use a color variable to accomplish the effect, as in:

```
01  my-color   PIC 9(5) VALUE ZERO.

01  my-field, ENTRY-FIELD, LINE 1, COL 1,
       COLOR my-color.

ADD 32768 TO my-color.
```

9. When the COLOR phrase is used with controls, each control type determines the allowable set of colors. Each control class defines its own method of assigning default colors. Note that many host systems limit the ability to define a control's colors. The colors used will be as close to the requested colors as the host system allows.

COLUMN NUMBER Phrase

```
AT {COLUMN  } NUMBER col-num
   {COL     }
   {POSITION}
   {POS     }
```

1. The COLUMN phrase specifies the terminal column to use. Column number one (1) indicates the leftmost column of the current subwindow. A column number of zero (0) indicates the current cursor column.

2. When the COLUMN phrase is used with controls and floating windows, the COLUMN NUMBER refers to the left edge of the control or window. You may use CELL(S) or PIXEL(S) to specify the coordinates of controls. With cells, non-integer values are allowed, and with pixels, non-integer values are not allowed.

3. If the COLUMN phrase is missing, the results depend on the compatibility mode being used.

   a. In VAX COBOL and ICOBOL compatibility modes, the current cursor position is used unless the LINE phrase is specified. In this case, column one is used. This rule is always applied when controls are placed, regardless of the compatibility mode being used.

   b. In RM/COBOL compatibility mode, column one is used for the first ACCEPT or DISPLAY operand. For subsequent operands, the current cursor position is used (implies COLUMN 0).

4. If the column number specifies a column value greater than the number of columns in the current window, then one is added to the line number used, and the column number is reduced by the width of the window. This is repeated until the column number specifies a value that is contained within the window.

   Note: This rule is not used when you are placing controls on the screen. If a control is placed outside of the subwindow, the control is not created.

   ---

5. The COLUMN NUMBER phrase used for entries in the Screen Section has a different format. See **section 5.9.5, "COLUMN Clause,"** for details.

CONTROL Phrase

```
CONTROL cntrl-string
```

1. The CONTROL phrase provides the ability to modify the static attributes of the ACCEPT or DISPLAY statement at runtime. The CONTROL data item is treated as a series of comma-separated keywords that control the action of the statement. Within the CONTROL data item, spaces are ignored and lower-case letters are treated as if they were upper-case. When a CONTROL item conflicts with the statically declared attributes of the ACCEPT or DISPLAY statement, the actions specified in the CONTROL item take precedence.

   The keywords allowed in *cntrl-string* are listed in the following groups. If more than one keyword from within a group appears in *cntrl-string*, only the rightmost one in the data item is used:

   - ERASE, ERASE EOL, ERASE EOS, NO ERASE.
   - BEEP, NO BEEP
   - HIGH, LOW, STANDARD, OFF
   - BLINK, NO BLINK
   - REVERSE, NO REVERSE
   - TAB, NO TAB

- PROMPT, NO PROMPT
- CONVERT, NO CONVERT
- UPDATE, NO UPDATE
- ECHO, NO ECHO
- UPPER, NO UPPER, LOWER, NO LOWER
- UNDERLINED, NO UNDERLINE
- LEFT, RIGHT, CENTERED, NO JUST
- SAME
- FCOLOR
- BCOLOR

2. Each of the keywords performs the same action as the statically declared attribute of the same name. The "NO" forms (NO ERASE, NO BLINK, etc.) cancel the effects of the named attribute.

   The FCOLOR and BCOLOR keywords are used to set foreground and background colors respectively. These keywords must be followed by an equals sign and the name of a color taken from the following list: BLACK, BLUE, GREEN, CYAN, RED, MAGENTA, BROWN, and WHITE. The named color becomes the default foreground or background color for the window. Note that this is different from the COLOR phrase, which sets the color only for the current ACCEPT or DISPLAY statement. The FCOLOR and BCOLOR keywords set the default colors for every subsequent ACCEPT and DISPLAY until explicitly changed.

CONVERT Phrase

```
WITH {CONVERSION}
     {CONVERT   }
```

1. Specifying the CONVERT phrase allows non-USAGE DISPLAY items to be accepted or displayed. It also allows for automatic conversion of entered data into the internal format of the destination field.

2. In a DISPLAY statement, the CONVERT phrase affects only numeric and numeric edited data items. For numeric data items, leading zeros are converted into spaces, a decimal point is inserted (if needed), and a leading minus sign is inserted (if the value is negative). The result is then left-justified if RM/COBOL compatibility mode is being used. Numeric edited items are not converted, but they are justified in the same manner as numeric items. This form of the CONVERT phrase is called output conversion.

   Floating-point data items are converted from internal format to E-notation. The format of E-notation is as follows:

   - FLOAT items have eight digits, one of which is before the decimal point.
   - DOUBLE items have 17 digits, one of which is before the decimal point.
   - A leading sign is shown for negative values, otherwise there is a leading space.
   - The exponent is shown as an “E”, followed by a sign if negative, followed by one or more digits.
   - The decimal point is shown using the program’s current decimal point character.

3. In an ACCEPT statement, the action of the CONVERT phrase depends on the data type of the receiving field:

   a. A nonnumeric field receives the entered data as if it were the destination of a MOVE statement with an alphanumeric source. This causes justification and editing to have their normal effect.

   b. A numeric or numeric edited field causes the ACCEPT statement to first check the data for correctness. Legal characters for numeric items include any characters that can appear in a numeric edited data item (digits, plus and minus sign, period, comma, currency symbol, “CR”, “DB”, “/”, and “*”). Characters that are legal but have no meaning in an arithmetic literal are simply ignored (space, “*”, “/”, currency symbol, currency comma). Any

other non-space character is an error. Legal values are then assigned to the receiving field such that the algebraic value remains the same.

c. The input format for floating-point numbers consists of the following components in the order given:

(1) An optionally signed string of digits (possibly containing a decimal point)

(2) An optional exponent field consisting of:

(a) "E" or "e"

(b) an optional + or -

(c) an integer

See **section 2.1.2.1, "Numeric literals,"** for additional information.

4. If the CONVERT phrase is not specified, the accepted data is placed in the receiving field from left to right with no editing or justification. In RM/COBOL compatibility mode, however, a numeric receiving field causes the data to be assigned from right to left instead.

5. For entries in the Screen Section, conversion is automatically used during input and not used during output. For this reason, Screen Section entries for non-integer numeric fields should specify a numeric edited PICTURE so that the decimal point is visible. Likewise, signed numeric fields should either specify SIGN SEPARATE or be represented by numeric edited items. Note that conversion occurs automatically for all controls that reference numeric, numeric edited, right justified, or wide character data.

6. If a conversion error occurs, the runtime system prints an error message and forces the user to correct the field. This behavior can be modified. For more information, see Book 1, *ACUCOBOL-GT User's Guide*, section 4.3.2.1 (subsection 3) "The KEYBOARD variable."

7. Several compile-time and run-time options are available to change various aspects of the CONVERT phrase, including the behavior of conversion errors and justification of the output of numeric items. For details, see Book 1, *ACUCOBOL-GT User's Guide*, **section 2.2.11**,

"Video Options" (specifically the "-Vc" and "-Ve" options). Also, see the Terminal Manager chapter of the *ACUCOBOL-GT User's Guide*, **section 4.4.2**, "The SCREEN Option."

DEFAULT Phrase

```
{DEFAULT [IS default]}
{UPDATE              }
```

1. The DEFAULT phrase allows for a default value to be displayed in an input field ready for editing by the user. It is displayed in a form that can be accepted with the CONVERT phrase. For numeric receiving fields, this will cause leading spaces to be removed. If a default value is specified, then it is the value displayed. Otherwise the current value of the receiving field is used. The UPDATE phrase is equivalent to the DEFAULT phrase with no default value specified.

2. If the PROMPT phrase is also specified, then the prompt character replaces any trailing spaces in the field.

3. The DEFAULT phrase is automatically implied for any update field specified in the Screen Section. The DEFAULT phrase implies the CONVERT phrase in a Format 1 ACCEPT statement.

Note: The "-Vu" compiler option allows you to imply the UPDATE phrase for all Format 1 ACCEPT statements that do not have an explicit UPDATE or DEFAULT phrase specified for them.

ECHO Phrase

```
ECHO
```

1. This causes the data entered to be redisplayed in the field. This allows for the action of the OUTPUT phrase to take place. Data will also be redisplayed with output conversion under the following circumstances:

   a. in VAX COBOL and ICOBOL compatibility modes, if the CONVERT or UPDATE phrase is used;

   b. in RM/COBOL compatibility mode, if the UPDATE phrase is used.

Note: NO ECHO is not the opposite of ECHO. See the NO ECHO phrase below for more details.

2. For entries specified in the Screen Section, the ECHO phrase is automatically implied. Output conversion does not occur, however.
3. See the CONVERT phrase above for details on the effects of output conversion.

ENABLED Phrase

```
ENABLED {IS} {TRUE         }
        {= } {FALSE        }
             {enabled-state}
```

The ENABLED phrase determines whether or not the control can be used. When a control is enabled, it can be used normally. When it is disabled, it can be seen on the screen (if it is also visible), but the user cannot interact with it. If the TRUE phrase is used, then the control is enabled. The FALSE phrase makes the control disabled. If enabled-state is used instead, then its value determines whether the control is enabled. Any non-zero value enables the control; a value of zero (0) disables it. If the ENABLED phrase is omitted, the control is initially created enabled.

Disabled controls are displayed differently than enabled controls. For example, under Windows, disabled controls usually have dimmed text. See the configuration variable **DISABLED_CONTROL_COLOR** in Appendix H for information on how to configure the appearance of a disabled control when running on a character-based system.

Note: Under Microsoft Windows, the appearance of disabled controls can change depending on whether the program is running under the 16- or 32-bit runtime. The differences are intrinsic to the Windows API and cannot be worked around.

ERASE Phrase

```
{ERASE} [TO END OF] {LINE  }     (VAX, ICOBOL)
{BLANK}             {SCREEN}
{ERASE} [EOS]                    (RM)
{BLANK} [EOL]
```

1. The ERASE phrase specifies that some or all of the current line or screen should be cleared to spaces. The spaces will use the currently selected background color for the active window, unless the "-Vi" compiler option was used (see Book 1, *ACUCOBOL-GT User's Guide*, section 2.1.10, "Video Options"). Cursor positioning occurs before the ERASE clause takes effect.

2. The ERASE LINE phrase causes the current line to be erased. The ERASE TO END OF LINE and ERASE EOL phrases both erase the line from the current cursor position to the end of the current window.

3. The ERASE and ERASE SCREEN phrases cause the current window to be cleared. The ERASE TO END OF SCREEN and ERASE EOS phrases cause the window to be erased from the current cursor position to the end of the window.

4. The cursor is not moved by these operations except in RM/COBOL compatibility mode, where ERASE (without the EOL or EOS phrases) specified on a Format 1 ACCEPT or DISPLAY verb causes the default cursor location to be line 1, column 1.

5. When ERASE SCREEN is specified in a Screen Section entry, and that entry either contains or is subordinate to a COLOR phrase, then the default window colors are changed to match that COLOR phrase. This also applies to the FOREGROUND-COLOR and BACKGROUND-COLOR phrases.

6. If a control has the TEMPORARY style and its upper-left location is within the erased area, the control will be destroyed. PERMANENT controls, however, are not affected.

EVENT-LIST, AX-EVENT-LIST, EXCLUDE-EVENT-LIST Phrases

```
EVENT-LIST     {IS} ( event-value { event-value ... } )
               {= }

AX-EVENT-LIST  {IS} ( ax-event-value { ax-event-value ... } )
               {= }

EXCLUDE-EVENT-LIST {IS} list-state
                   {= }
```

1. These phrases provide a mechanism for specifying a list of event types to either send or withhold (block) from the program depending on the value of EXCLUDE-EVENT-LIST. By default, listed events are sent to the program.

2. *Event-value* and *ax-event-value* are numeric literals or data items that identify an event type. List elements must be enclosed by parentheses and separated by a space. If the list contains a single element, the parentheses can be omitted.

3. *List-state* is an integer literal or numeric data item. Valid values are "0" and "1".

4. EVENT-LIST and AX-EVENT-LIST hold a list of numeric values that correspond to event types.

5. EVENT-LIST is used with ACUCOBOL-GT graphical controls. See Book 2, Chapter 6, "Events Reference" for a discussion of events related to graphical controls, including a description of each event and its value.

6. AX-EVENT-LIST is used with ActiveX and .NET controls. ActiveX and .NET events associated with a given control are listed in the COPY file (".def") created for that control with the AXDEFGEN or NETDEFGEN utility.

7. The value of EXCLUDE-EVENT-LIST determines whether the event types in EVENT-LIST and AX-EVENT-LIST are sent to or withheld from the program. When EXCLUDE-EVENT-LIST is set to "0", the default value, events in the list are sent to the program. When EXCLUDE-EVENT-LIST is set to "1", events in the list are blocked (not sent to the program).

8. Three runtime configuration variables, **TC_EVENT_LIST**, **TC_AX_EVENT_LIST**, and **TC_EXCLUDE_EVENT_LIST**, augment this facility in thin client deployments. For more information, see Chapter 6 of the *AcuConnect User's Guide*.

FONT Phrase

```
FONT {IS} font-handle
     {= }
```

Font-handle identifies the font to use for the control. If the font phrase is omitted, the window's default font for controls is used.

If you change a control's font after the control has been created, and its size is based on its font (true for controls that do *not* use the CELLS phrase), the control's dimensions are recomputed using the new font.

FOREGROUND-COLOR and BACKGROUND-COLOR Phrases

```
{FOREGROUND-COLOR } IS fg-color
{FOREGROUND-COLOUR}

{BACKGROUND-COLOR } IS bg-color
{BACKGROUND-COLOUR}
```

1. These phrases allow the specification of the foreground or background color. The color is a literal value taken from the following table:

Value	Color
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	White
8	Dark-Gray
9	Bright-Blue
10	Bright-Green
11	Bright-Cyan
12	Bright-Red
13	Bright-Magenta
14	Yellow

Note: The values in this table are not the same as the Background Color and Foreground Color values in the "acucobol.def" file.

2. *Fg-color* and *bg-color* can also be used to set a control's colors . They specify both foreground and background colors in the same manner as they do for textual screen fields.

   Each control type determines the allowable set of colors and the interpretation of *foreground color* and *background color.* If either the foreground color or background color is omitted (or its particular value is not meaningful), then default colors are used. Each control class defines its own method of assigning default colors. Note that many host systems limit the ability to define a control's colors. The colors used will be as close to the requested colors as the host system allows.

3. If both the COLOR phrase and the FOREGROUND-COLOR or BACKGROUND-COLOR phrase appear in the statement, the COLOR phrase takes precedence.

4. These phrases are provided for compatibility with other COBOL systems. Note that the color value must be an integer literal or a numeric data item, and it may be an arithmetic expression. Also note that the color number is one less than the corresponding color value in the COLOR phrase (this is done to maintain compatibility with other COBOLs). For a method of assigning colors that can be variable, see the COLOR phrase above.

Note: The "-Vi" compiler option changes the behavior of a Screen Section ERASE (and BLANK) phrase with respect to color handling. See **Section 2.2.11, "Video Options"**of the *ACUCOBOL-GT User's Guide* for details.

FULL Phrase

```
{FULL         }
{LENGTH-CHECK}
```

1. The FULL phrase applies only to input and update fields. When it is specified, the user must either leave the field blank or must enter non-space data into both the first and last character positions of the screen field. The user will not be able to leave the field until these requirements are met.

2. If the user types an exception key (and exception keys are allowed), then the ACCEPT will terminate without forcing the user to meet the requirements of the FULL phrase.

HELP-ID Phrase

```
HELP-ID {IS} help-id
        {= }
```

The HELP-ID phrase establishes *help-id* as the control's help ID. The help ID is used in conjunction with certain events to provide context-sensitive application help. Usually, each control is given a unique, system-wide value so that it can be easily identified by the help processor. The maximum value of *help-id* is 2,147,483,647. For more information about HELP-ID and help automation, see Chapter 10, Book 2, *User Interface Programming*.

HIGH, LOW, and STANDARD Phrases

```
{HIGHLIGHT}
{HIGH     }
{BOLD     }
{LOWLIGHT }
{LOW      }
{STANDARD }
```

1. The HIGH, LOW, and STANDARD phrases set the high/low video attributes for the input field. If the terminal hardware does not support high/low intensity, then these phrases are ignored.

2. HIGH sets the video attribute to high intensity. LOW sets the video attribute for the field to low intensity.

3. STANDARD sets the video attribute for the field to be the default intensity for the terminal type being used. This will either be high or low intensity depending on the terminal.

4. If none of these phrases is specified, STANDARD is used. This may be overridden by compile time options to set the default to either HIGH or LOW. For details, see Book 1, *ACUCOBOL-GT User's Guide*, section 2.1.10, "Video Options."

IDENTIFICATION Phrase

```
{IDENTIFICATION} {IS} control-id
{ID            } {= }
```

1. Use *control-id* to assign a unique identifier to a control. This identifier is used any time that control must communicate with your program. It is returned with any messages from the control. You can examine the ID to distinguish which control sent the message. This can be more convenient than using the control's handle because you can assign a fixed identifier that is the same in every run of the program.

2. If *control-id* is omitted, then it will be treated as if it were zero (0), unless the control is defined in the Screen Section. In this case it will be assigned the same value as the control's *field number* (see **section 5.9**, General Rules for Format 2, for a description of how Screen Section field numbers are assigned). Control IDs must be in the range of 0 to 32767.

KEY Phrase

```
KEY {IS} key-letter
    {= }
```

1. The KEY phrase assigns a *key letter* to the control. The user can activate the control by typing its key letter in combination with some other special key. Under Microsoft Windows, key letters are typed in conjunction with the "Alt" key. The first character in *key-letter* becomes the control's key letter. The key letter can be given in lower- or upper-case.

2. If the KEY phrase is omitted, the control's key letter is derived from its title. (See the TITLE phrase for details.) If a key letter is not indicated in the title, the control does not have a key letter. If a key letter is designated in both the title and the KEY phrase, the KEY phrase takes precedence. If the letter specified in the KEY phrase is not the same as

the letter indicated in the title, the letter indicated in the title is highlighted and the letter specified in the KEY phrase is the key letter (this could be confusing to the user).

3. The activation technique for key letters is shared with the menu bar handler on most systems. You should avoid assigning duplicate key letters in any one floating window, or assigning key letters that conflict with key letters used by the window's menu bar.

4. Controls that cannot be activated, such as a label, may have key letters (see the example that follows). In this case, the runtime system uses the following rules to determine which control to activate when the key letter is typed:

   a. if another control has the same key letter, then the first control created with that key letter is activated when that key letter is pressed; otherwise,

   b. the next control that can be activated (in the same floating window) is activated. The runtime uses the order in which the controls were created to determine which control is next.

   In practice, these rules mean that you can usually assign a key letter in the title of a label and use rule b) to associate that key letter with a corresponding entry field. In this case, you do not need to use the KEY phrase. If rule (b) does not give you the effect you desire, you can use the KEY phrase and rule (a) to explicitly state the effect you want.

   For example, the following Screen Section segment:

```
03 LABEL "&Customer: ".
03 ENTRY-FIELD USING CUST-NO, COLUMN + 2."
```

   would assign a key letter of "C" to the label (using the implied key letter assignment from the label's title). Since labels cannot be activated, typing Alt-C would activate the CUST-NO entry field because it is the next control that can be activated.

Note: If you use "&" in the control title, the letter that directly follows it becomes the accelerator key for this control, indicated by an underline. If you use the KEY IS construct to specify the accelerator key, that letter is not underlined in the control's title.

---

5. A control's key letter is assigned when the control is created. It cannot be changed.

LAYOUT-DATA Phrase

```
LAYOUT-DATA {IS} layout-data
            {= }
```

The LAYOUT-DATA phrase sets the control's LAYOUT-DATA property. Some layout managers use this property to determine how to manage certain attributes of the control, such size and position. See the documentation for the specific layout manager for a description of *layout-data* and its accepted values. Layout managers are documented in Book 2, section 4.8, "Layout Managers."

LINE NUMBER Phrase

```
AT LINE NUMBER line-num
```

1. The LINE NUMBER phrase is used to position the cursor on a specific line or *cell row* of the terminal. A line number value of one indicates the top line of the current subwindow. To support fine positioning of controls in graphical environments, non-integer values are allowed. On text-mode systems, non-integer values are rounded down to the nearest whole number.

2. When used with controls and floating windows, the line number positions the top edge of the control or window.

3. If the LINE NUMBER phrase is missing or zero, then the results depend on the compatibility mode being used.

   a. In VAX COBOL and ICOBOL compatibility modes, the current cursor line is used. This is used in all cases when the statement refers to controls.

   b. In RM/COBOL compatibility mode, the current line is used if the COLUMN phrase specifies a value of zero or the NO ADVANCING phrase is used. Otherwise, the next line is used. In any case, if ERASE is also specified (without EOL or EOS), then line 1 is used instead.

4. If the line number results in a value greater than the number of lines in the current window, the current window is scrolled up one line, and the line number is set to be the bottom line of the current window.

   This rule does not apply to controls. If a control is positioned outside of the current subwindow, it is not created.

5. You may use LINE NUMBER with different units, such as CELLS or PIXELS. PIXELS apply only with controls, and not with windows. Non-integer values are allowed with cells, but they are not allowed with pixels.

6. The LINE NUMBER phrase used for entries in the Screen Section has a different format. See **section 5.9.4, "LINE Clause,"** for details.

LINES Phrase

```
LINES {IS} height
      {= }
```

1. The LINES phrase is used to set a window or control's height. Note that each control type defines exactly how this value is interpreted.

   Typically, the *height* field represents the logical height of the control. Frequently, this is based on the size of the control's title or value. For example, a LABEL control's height of "1" would specify a one-line label. If this phrase is omitted, the control type employs its own method of determining a default size.

2. The CELLS option of the LINES phrase causes the HEIGHT-IN-CELLS control style to be used. This means that the height of the control is specified exactly using the cell size of the owning window. For example:

   ```
   LINES = 15 CELLS
   ```

   causes the control to have a height of 15 cells, where the exact height of a cell is inherited from the parent window.

3. You may employ the PIXELS option with the LINES phrase to specify the control's size more precisely using the pixel size of the owning window. For example:

   ```
   LINES = 60 PIXELS
   ```

MAX-HEIGHT, MAX-WIDTH, MIN-HEIGHT, MIN-WIDTH Phrases

```
MAX-HEIGHT {IS} max-height
           {= }
MAX-WIDTH  {IS} max-width
           {= }
MIN-HEIGHT {IS} min-height
           {= }
MIN-WIDTH  {IS} min-width
           {= }
```

1. MAX-HEIGHT, MAX-WIDTH, MIN-HEIGHT, and MIN-WIDTH are common properties of all graphical controls. They are unique from other common properties in that their names are reserved by the compiler only when they appear in the context of acting on a control: in a Screen

Section entry, or a DISPLAY, MODIFY, or INQUIRE statement. (For more about how the compiler reserves style and property names, see section 4.4, "Styles and Special Properties," in Book 2, *ACUCOBOL-GT User Interface Programming*.)

2. The MAX-HEIGHT, MAX-WIDTH, MIN-HEIGHT, MIN-WIDTH phrases are used to specify a control's maximum and minimum height and width. These values are used by layout managers when resizing a control (see section 4.8, "Layout Managers," in Book 2, *ACUCOBOL-GT User Interface Programming*).

3. *max-height* is the maximum height of the control. It's value is in the same units as the control's LINES setting. The default value, zero, indicates no maximum height.

4. *max-width* is the maximum width of the control. It's value is in the same units as the control's SIZE setting. The default value, zero, indicates no maximum width.

5. *min-height* is the minimum height of the control. It's value is in the same units at the control's LINES setting. The default value, zero, indicates no minimum height.

6. *min-width* is the minimum width of the control. It's value is in the same units at the control's SIZE setting. The default value, zero, indicates no minimum width.

NO ADVANCING Phrase

```
WITH NO ADVANCING
```

1. This phrase inhibits the normal carriage-return, line-feed sequence that is sent to the terminal. In VAX COBOL and ICOBOL compatibility modes, this sequence is normally sent *after* the ACCEPT or DISPLAY statement executes. In RM/COBOL mode, this sequence is sent *before* the statement executes. These characters can cause the current terminal window to scroll if the current line is the last line in the window.

2. This phrase is automatically implied when the LINE phrase is used. In VAX COBOL compatibility mode, this phrase is also implied when the COLUMN phrase is used. In RM/COBOL mode, this phrase is also implied when the COLUMN phrase is used with a zero value.

NO ECHO Phrase

```
{WITH NO ECHO}
{NO-ECHO     }
{SECURE      }
{OFF         }
```

The NO ECHO phrase enables data to be entered without being displayed on the terminal screen. This can be used to enter passwords and other secure data.

OUTPUT Phrase

```
OUTPUT {JUSTIFIED}  {LEFT    }
       {JUST     }  {RIGHT   }
                    {CENTERED}
```

1. The OUTPUT phrase affects the alignment of displayed data. It has no effect in a Format 1 ACCEPT statement unless the ECHO phrase is also specified.

2. The LEFT option causes data to be justified to the left edge of the field, the RIGHT option causes right-justification, and the CENTERED option causes the displayed data to be centered in the field.

3. When the OUTPUT phrase is used, leading and trailing spaces are removed from the data before it is displayed. This is also done before the default size of the field is computed (except for Screen Section entries). Thus the three forms of justification are the same unless the SIZE phrase is also specified with a size larger than the data's length. Computing the default size after stripping the leading and trailing spaces allows fields to be concatenated together on the screen without computing column positions.

PROMPT Phrase

```
PROMPT [ CHARACTER IS prompt-lit ]
```

1. The PROMPT phrase causes the input field to be filled with the specified prompt character prior to entry. If the prompt character is omitted, underscores are used.

2. If the DEFAULT phrase is also used, then the prompt character replaces any trailing spaces in the field.

3. The PROMPT phrase is automatically used for every entry in the Screen Section. Thus this phrase is treated as commentary in the Screen Section unless it specifies a prompt character other than underscores.

PROPERTY and Property-Name Phrases

```
{ property-name          } {IS }
   { prop-option [GIVING result-1] }...
{ PROPERTY property-type } {ARE}
{ method-name            } {=  }
{ object-expression      }
```

where *prop-option* is one of the following:

```
{ property-value [ LENGTH {IS} length-1 ] }
{                         {= }            }
{                                         )
{ ( {property-value} ... )                }
{                                         }
{ { MULTIPLE } property-table             }
{ { TABLE    }                            }
{                                         }
{ parameter                               }
{                                         }
{ ( { parameter } ... )                   }
```

1. The PROPERTY phrase assigns a value to one of a control's *special* properties or invokes a control-specific *method*. The PROPERTY phrase takes the types of only special properties and methods for controls (for a discussion of Common and Special properties, see section 5.1, Book 2, *ACUCOBOL-GT User Interface Programming;* for a discussion of Methods, see section 4.5 of that book). Each type of control has its own set of special properties and methods. These are described in the sections documenting each control type. *Property-type* specifies which special property to modify or which method to invoke (each of a control's special properties and methods is uniquely identified by a number). *Property-value* is the value to assign to that special property. *Property-value* must be a data type that is appropriate for the specified property. If *property-type* specifies a special property that does not exist, it is ignored for most control types. If the control type is ACTIVE-X, then an exception is raised (for a discussion of the ActiveX

control types, see section 5.3, Book 2, *ACUCOBOL-GT User Interface Programming*). Each property's unique value is defined in the file "controls.def" or in the ActiveX control's COPY file.

2. *Property-name* and *method-name* provide an alternate method for identifying which special property to modify or which method to invoke. The compiler knows the names of the special properties and methods that belong to each control type. In situations where the compiler knows which type of control is being acted upon, you can use the appropriate *property-name* or *method-name* directly instead of using the special property's or method's identifying number in the PROPERTY phrase.

   For example, the MAX-TEXT special property of entry fields is property number "1". You can set the value of this property to "10" with either of the following phrases:

   ```
   PROPERTY 1 = 10
   MAX-TEXT = 10
   ```

   The second method can be used only when your code makes it clear to the compiler that you're acting on an entry field.

   You can use either the PROPERTY phrase or *method-name* to specify which method to invoke. For example, the LoadFile method of the Microsoft Rich Textbox Control is method number "37". You can invoke this method with either of the following phrases:

   ```
   PROPERTY 37 ("myfile.rtf", rtfRtf)
   LoadFile ("myfile.rtf", rtfRtf)
   ```

   The second method can be used only when your code makes it clear to the compiler that you're acting on a Microsoft Rich Textbox Control.

3. Some properties return specific values when set. These values are placed in *result-1* of the GIVING phrase. The meaning of the value depends on the property being set; see the documentation for the specific property. Properties that do not have a pre-defined return value set *result-1* to "1" if the property was set successfully, or "0" if not. When a property is being given multiple values in a single assignment (for example: "Display-Columns = (1, 10, 30)"), then *result-1* is set in response to the last value assigned.

4. When multiple special property assignments are made in a single statement, those assignments are performed in the order listed in the statement.

5. If more than one *property-value* is specified, each one is applied to the property in the order listed. This is normally used for *cumulative* properties. These are properties that perform some special action each time they have a value assigned to them. For example, you can set three columns in a list box with "DISPLAY-COLUMNS = (1, 20, 35)". In the case of the DISPLAY-COLUMNS property, each time it is assigned a value, it sets a new column location. Note that the parentheses are required.

   When you specify *property-table*, then each element of the table is assigned to the property. The elements are assigned in ascending occurrence order. For example, the following code fragment fills a list box with the names of three colors:

```
01  COLOR-NAMES.
    03  PIC X(10) VALUE "Red".
    03  PIC X(10) VALUE "Green".
    03  PIC X(10) VALUE "Blue".

01  COLOR-TABLE REDEFINES COLOR-NAMES
    OCCURS 3 TIMES
    PIC X(10).

PROCEDURE DIVISION.
MAIN-LOGIC.
    DISPLAY LIST-BOX, SIZE 10, LINES 3,
            ITEM-TO-ADD = TABLE COLOR-TABLE.
```

Note: The current size of the table is used, so you can use OCCURS DEPENDING ON tables when you want to have a variable number of items in a table.

You should use caution when specifying property tables in the Screen Section. Remember that each DISPLAY statement of a Screen Section item reloads all of that item's properties into the control. This can be inefficient if the property table is large, and it can cause duplicate entries if you are not careful. To avoid this, you can create your controls in the Screen Section, but use the MODIFY statement to set any table-oriented

properties at the appropriate point in your program. In this way, the tables are not referenced in the Screen Section and a DISPLAY will not cause those tables to be reprocessed.

6. When the LENGTH option is specified, *length-1* establishes the exact size of *property-value*. The text value presented to the control must not contain trailing spaces or have trailing spaces added. When you specify the LENGTH option, the control uses exactly the number of characters of *length-1*. However, if *length-1* is a value larger than the size of the data item it is modifying, the size of the data item is used instead. If *length-1* is negative, it is ignored and the default handling occurs.

7. {*parameter*}... is a list of parameters to pass when invoking one of a control's methods or setting a multiple-parameter property. When setting a multiple-parameter property, the first parameters identify which aspect of the property to set. The last parameter is the actual property value. For example, to set the Microsoft Chart Control DataGrid::RowLabel property you must specify the row number and label index. You could use the following phrase:

   ```
   DataGrid::RowLabel( ROW-NUMBER, ROW-LABEL-INDEX,
       "My Row Label" )
   ```

8. *Object-expression* can only be used in the procedure division, not the screen section. It has the following format:

   ```
   { {^} property-1 [ ( param-1 ... ) ]
     [ :: property-2 [ ( param-2 ... ) ] ]... }
   ```

   Object-expression specifies a property or method of an object referenced by another object. This object in turn can be referenced by yet another object. The "root" object can be an ActiveX control or COM object or a graphical control. "^" can only be used in conjunction with Format 5 USE verb (see the documentation for the USE verb for more information). *Property-1* is the name of a property of the ActiveX control or COM object. *Property-1* must not be a write-only property. *Property-2* is the name of a property of the ActiveX control or COM object which is the value of *property-1*. *Property-2* must not be a write-only property. *Param-1* and *param-2* are literals, data items or numeric expressions. *Param-1* is the first parameter passed when getting the value of *property-1* and *param-2* is the first parameter passed when getting the value of *property-2*.

For example, to set the Microsoft Chart Control legend, you get the value of the Legend property. This value is an object that you may then modify to change the legend. The Legend object has properties whose values are other objects, and so on. The following phrases set properties and invoke methods of the Microsoft Chart Legend object:

```
Legend::Location::Visible = 1
Legend::Location::LocationType = VtChLocationTypeRight
Legend::TextLayout::HorzAlignment =
          VtHorizontalAlignmentRight
Legend::VtFont::VtColor::Set (255, 255, 0)
Legend::BackDrop::Fill::Style = VtFillStyleBrush
Legend::BackDrop::Fill::Brush::Style = VtBrushStyleSolid
Legend::BackDrop::Fill::Brush::FillColor::Set (255, 0, 255)
```

or assuming the handle to the Microsoft Chart Control is MS-CHART-1:

```
USE MS-CHART-1 Legend
      MODIFY ^Location::Visible = 1
         ^Location::LocationType = VtChLocationTypeRight
         ^TextLayout::HorzAlignment =
               VtHorizontalAlignmentRight
         ^VtFont::VtColor::Set ( 255, 255, 0 ).
USE MS-CHART-1 Legend::BackDrop::Fill
      MODIFY ^Style = VtFillStyleBrush
         ^Brush::Style = VtBrushStyleSolid
         ^Brush::FillColor::Set ( 255, 0, 255 ).
```

REQUIRED Phrase

```
{REQUIRED   }
{EMPTY-CHECK}
```

1. The REQUIRED phrase applies only to input and update fields. When it is specified, the user may not leave the field empty--some non-blank data must be entered. If the receiving field is numeric or numeric edited (and input conversion is being used), then the entered value may not be zero. The user will not be able to leave the field without meeting these requirements.

2. In a Screen Section, the REQUIRED phrase forces the user to enter data in the field *before terminating the ACCEPT.*

This prevents the user from jumping to the last field with the mouse, and thus bypassing a required field. If the user attempts to terminate an ACCEPT while required fields are still blank, an error message is displayed and the cursor is positioned at the first required field that is blank.

3. If the user types an exception key (and exception keys are allowed), the ACCEPT will terminate *without* forcing the user to meet the requirements of this phrase. Note that leaving a field by clicking on another control with the mouse will cause the REQUIRED check to occur. Therefore, "CANCEL" buttons should always have the SELF-ACT style. The SELF-ACT style does not cause the runtime to change the focus of the current control - the input or update field is not left - and the REQUIRED phrase is not triggered. Without the SELF-ACT style, the CANCEL button will cause focus to shift to itself, triggering the REQUIRED check and preventing the user from "canceling" without first entering some data.

REVERSED Phrase

```
{REVERSE-VIDEO}
{REVERSE      }
{REVERSED     }
```

The REVERSED phrase causes the field to have reversed foreground and background colors. If the terminal hardware does not support reverse-video, then this phrase is ignored.

SAME Phrase

```
SAME
```

1. The SAME phrase causes the field to use the video attributes currently present at the field's screen location. This allows data on the screen to be changed without changing the screen's video attributes.

2. The SAME phrase may not be specified along with the UNDERLINED, BLINK, REVERSED, HIGH, LOW, STANDARD, COLOR, FOREGROUND-COLOR, or BACKGROUND-COLOR phrases.

SCROLL Phrase

```
SCROLL [UP  ] [ BY scrl-num {LINE } ]
```

```
[DOWN]              {LINES}
```

1. The SCROLL phrase specifies the number of lines to scroll the current window. Scrolling is the first operation performed by the ACCEPT or DISPLAY statement--it occurs before any cursor positioning or the operation of any ERASE clauses. Scrolling does not affect the current cursor position.

2. If the UP phrase is specified, the contents of the current window are scrolled upward (normal scrolling). If DOWN is used, then the contents are moved downward (reverse scrolling). A blank line with the default video attributes for the window is introduced at the top or bottom of the window (as appropriate). Upward scrolling is used if neither the UP nor DOWN phrase is specified.

3. If a scrolling value is specified, its value is the number of lines to scroll the screen. If this value is not positive, no scrolling occurs. If the scrolling value is omitted, one line is scrolled.

SIZE Phrase (with a text entry field)

```
WITH PROTECTED SIZE length
```

1. The SIZE phrase specifies the size (length) of the field. After the ACCEPT or DISPLAY is finished, the cursor is placed immediately after the field defined by this clause, unless this would place the cursor outside of the current terminal window. In this case, the cursor is wrapped around to the beginning of the next line (scrolling the window if necessary).

2. If the SIZE phrase is not used, then the field length defaults to the size of the item being accepted or displayed. If the CONVERT phrase is used, however, then the size of the field depends on the data type of the item and the verb being used:

    a. If the DISPLAY verb is executing, then the size is the same as if the CONVERT phrase were not specified *except* for numeric items. For numeric items, the size is the number of digits in the item, plus one if it is not an integer, plus one if it is signed. This rule also applies to the action of the ECHO phrase of the ACCEPT statement (see below). The remaining cases cover the size when an ACCEPT statement is used.

b. If the item is numeric or numeric edited, then the size is the number of digits in the item, plus one if it is not an integer, plus one if it is signed.

c. If the item is alphanumeric edited, then the size is set to the number of "A" or "X" positions specified in its PICTURE clause.

d. For all other data types, the field size is set to the size of the item (same as if CONVERT were not specified).

Note: You cannot supply the CONVERT phrase in the Screen Section. Thus the size of a Screen Section field is always the size of its screen entry unless the SIZE phrase is specified.

---

3. Note that the OUTPUT phrase changes the way in which the default field size is computed. See that heading above for details. Also note that the OUTPUT phrase affects only the way items are displayed on the screen; the internal format of accepted data is not affected.

SIZE Phrase (with Windows and Controls)

```
SIZE {IS} length [CELL  ]
     {= }        [CELLS ]
                 [PIXEL ]
                 [PIXELS]
```

1. The SIZE phrase is used to set a window or control's width (length). Non-integer values are allowed. Note that each control type defines exactly how this value is interpreted.

   Typically, the *length* field represents the logical width of the control. Frequently, this is based on the size of the control's title or value. For example, an ENTRY-FIELD's width of "8" would specify a field large enough to enter eight (average size) characters. If this phrase is omitted, the control type employs its own method of determining a default size.

2. The CELLS option of the SIZE phrase causes the WIDTH-IN-CELLS control style to be used. This means that the width of the control is specified exactly using the cell size of the owning window. For example:

   ```
   SIZE = 15 CELLS
   ```

causes the control to have a width of 15 cells, where the exact width of a cell is inherited from the parent window.

3. You may use the SIZE phrase with PIXELS when defining controls. This means that the width of the control is specified exactly using the pixel size of the owning window. For example:

   ```
   SIZE = 150 PIXELS
   ```

   causes the control to have a width of 150 pixels, where the exact width of a pixel is inherited from the parent window. Non-integer values are not allowed with pixels.

STYLE Phrase and Style-Name

```
STYLE {IS} style-flags
      {= }

{style-name} ...
```

1. In the STYLE phrase, *style-flags* is a numeric field that holds a value that specifies the styles to apply to the control. Each control type defines its own set of styles and how the *style-flags* value is interpreted. *Style-flags* holds the sum of the numbers that represent the desired styles. Each style's identifying number is defined in the file "controls.def". If *style-flags* is omitted, the default style attributes are applied to the control.

2. A *style-name* is the name of a valid style for the type of control being acted upon. For example, some of the styles that apply to a radio-button include: BITMAP, FRAMED, and NOTIFY. Each *style-name* causes that style to be applied to the control.

3. You may use both the STYLE phrase and individual *style-names* for a particular control. The effect is to add the set of specified styles together. You would typically use the STYLE phrase to specify styles that may change at runtime, and *style-name* for those styles that are fixed.

   For more information about control styles, and the STYLE and style-name phrases, see section 4.4, Book 2, *ACUCOBOL-GT User Interface Programming*.

TAB Phrase

```
TAB
```

The TAB phrase forces the user to finish entering data by typing a valid termination key. It is the opposite of the AUTO phrase. The TAB phrase is the default behavior except in a Format 1 ACCEPT statement when you are using RM/COBOL compatibility mode.

TITLE Phrase

```
TITLE {IS} title
      {= }
```

1. The value of *title* sets the control's title. If *title* is omitted, then the control does not have a title. Some controls (such as entry fields, list boxes, and combo boxes) do not have titles. In these cases, any specified title is ignored.

2. If you change the title of a control and that control was created with the default sizing rules (did not have a size explicitly specified for it), the control is resized based on the length of the new title.

3. You can specify a *key letter* in a title by including the "&" character immediately before the key letter. The key letter is usually highlighted by the host system (under Windows, the key letter is underlined). The "&" character both defines the key letter and performs the highlighting. The key letter indicates that the user can go to the appropriate field by typing the key letter in combination with another key. Exactly how this occurs is system dependent. Under Windows, the user presses the "Alt" key in conjunction with the key letter. Some systems do not support key letters, in which case the first "&" character in a title is ignored. When the KEY phrase is used and an "&" character appears in *title*, the KEY phrase specifies the key letter and the character following the "&" indicates the highlighted character.

UNDERLINED Phrase

```
{UNDERLINE }
{UNDERLINED}
```

The UNDERLINED phrase causes the field to have the underscore video attribute applied to it. If the terminal hardware does not support underlining, then this phrase is ignored. UNDERLINED cannot be combined with BLINK.

UPON Phrase

```
UPON new-window
```

1. The UPON phrase is used to update a floating window that is not the *current* floating window. In this phrase *new-window* is a USAGE HANDLE or PIC X(10) data item that contains a valid floating window handle. The floating window specified by *new-window* becomes the *current* window for the duration of the DISPLAY statement.

2. If you create a new floating window while in the scope of the UPON phrase, the new window becomes the current window after the DISPLAY statement terminates. Otherwise, the window that was current before the DISPLAY UPON statement executed is restored to the current window. For example, if the main application window is current and you execute:

   ```
   DISPLAY FLOATING WINDOW UPON MAIN-WINDOW, ...
   ```

   when the DISPLAY statement terminates, the new floating window becomes the current window, instead of the main application window.

Note: *new-window* is always the current window for the entire remaining DISPLAY statement, even if you mix DISPLAY formats. For example:

```
DISPLAY SUBWINDOW UPON WINDOW-1,
   AT 0504, LINES 5, SIZE 30, BOXED
   "Line 1", LINE 1,
   "Line 2", LINE 2.
```

The above code creates a new subwindow in WINDOW-1 and then displays two lines in the new subwindow. The UPON WINDOW-1 phrase applies to both the DISPLAY SUBWINDOW operation and the display of the subsequent text items because they are all specified in one DISPLAY statement.

3. When you mix several DISPLAY formats, the logic to determine the current window is applied sequentially through the statement. Here is a complex example:

```
DISPLAY FLOATING WINDOW UPON WINDOW-1
   LINES 10, SIZE 40, BOXED,
   HANDLE IN WINDOW-99;
   "Line 1", LINE 1,
   "Line 2", UPON WINDOW-2, LINE 2
   "Line 3", LINE 3.
```

In this example, the new floating window (WINDOW-99) is created with WINDOW-1 as its parent (because of the first UPON phrase). Normally, this UPON phrase would cause "Line 1" to display in WINDOW-1 too, but the DISPLAY FLOATING WINDOW operation causes the new window to become the current window. So, "Line 1" is shown in WINDOW-99 instead. This would apply to "Line 2" also, but it specifies its own UPON phrase, so it displays in WINDOW-2. "Line 3" also displays in WINDOW-2 because that was the last window specified. At the end of the DISPLAY statement, the new floating window, WINDOW-99, is made the current window.

UPPER and LOWER Phrases

```
{UPPER}
{LOWER}
```

When the UPPER phrase is specified, all lower-case characters received by the ACCEPT statement are converted to upper-case. If the LOWER phrase is specified, all upper-case characters are converted to lower-case. If the ECHO phrase is also specified, the redisplayed data will be in the converted form.

Note: It may be necessary for you to use the configuration variable **UPPER_LOWER_MAP** to ensure correct translation.

VALUE Phrase

```
VALUE {IS} [ MULTIPLE ] value
      {= } [ TABLE    ]
```

1. *Value* sets the value of the control. If the control does not take a value, then the VALUE phrase is ignored. The exact interpretation of *value* depends on the type of control. A control's value is either an integer or

an alphanumeric string. If the control takes an integer value, then *value* should specify a numeric literal or data item. If the control takes an alphanumeric string, then *value* may be either numeric or alphanumeric. If *value* is numeric, it is converted to an alphanumeric string that represents its numeric value.

2. The MULTIPLE phrase is used only with certain control types that allow for multiple values. Its use is described in the sections that discuss those controls. The Screen Section VALUE phrase has some additional rules. See **section 5.9** for details. TABLE is a synonym for MULTIPLE.

VISIBLE Phrase

```
VISIBLE {IS} {TRUE         }
        {= } {FALSE        }
             {visible-state}
```

The VISIBLE phrase determines whether or not the control is shown on the screen. If the VISIBLE phrase is set to TRUE, the control is shown. If it is set to FALSE, the control is invisible. If *visible-state* is used instead, then its value determines whether the control is shown. Any non-zero value indicates that the control is visible; zero indicates that it is invisible. Controls that are invisible do not appear on the screen and cannot be used. They continue to exist, however, and can be made visible subsequently in the program. If the VISIBLE phrase is omitted, then the control is initially made visible.

ZERO-FILL and NUMERIC-FILL Phrases

```
{ZERO-FILL   }
{NUMERIC-FILL}
```

1. The ZERO-FILL and NUMERIC-FILL phrases may be applied only to alphanumeric input or update fields. These phrases are ignored for output fields. Only one of these options may be specified for a single field.

2. The ZERO-FILL phrase causes trailing spaces in the entered data to be replaced by the “0” character. If the destination is JUSTIFIED, then leading spaces will be replaced instead.

3. The NUMERIC-FILL phrase causes the runtime system to examine the entered data. If the entered value consists entirely of digits and trailing spaces, then the entered digits are right justified and the leading

character positions are replaced by the “0” character. If the entered data is empty, or contains any non-digit characters, the data is left alone.

6.5 Procedure Division Format

General Format

Format 1

```
PROCEDURE DIVISION

   [ {USING   } {parameter} ... ] .
     {CHAINING}

[ DECLARATIVES.

{ section-name SECTION [segment-no] .

   declarative-sentence

[ paragraph-name.

[ sentence ] ... ] ... } ...

  END DECLARATIVES. ]

{ section-name SECTION [segment-no] .

[ paragraph-name.

   [ sentence ] ... ] ... } ...
```

Format 2

```
PROCEDURE DIVISION

   [ {USING   } {parameter} ... ] .
     {CHAINING}

{ paragraph-name.
```

```
[ sentence ] ... ] ... } ...
```

Syntax Rules

1. If one paragraph is in a section, all paragraphs must be in sections.
2. A *section* consists of a section header followed by zero or more paragraphs.
3. A *paragraph* consists of a paragraph header followed by zero or more sentences.
4. A *sentence* is one or more statements terminated by a period.
5. *Parameter* must refer to a data item defined with a level-number of 01 or 77. If the USING phrase is used, then each *parameter* must be declared in the Linkage Section. If the CHAINING phrase is used, then each *parameter* must be declared in the File or Working-Storage Section.
6. Each *parameter* may not appear more than once in a USING/ CHAINING phrase.
7. *Declarative-sentence* must be a USE statement followed by a period.
8. *Segment-no* must be a number between zero and 99.
9. Each *segment-no* in the Declaratives must be less than 50.
10. If the *segment-no* is absent, it is treated as if it were zero.
11. Each *section-name* must be a unique user-defined word (exception: a *section-name* may be the same as a data name).

General Rules

1. The USING phrase is used only for programs that are invoked by a CALL statement with a USING phrase.
2. The USING phrase identifies the names used in the program to refer to arguments passed from the calling program. In the calling program, the USING phrase of the CALL statement identifies the arguments. The two USING lists correspond positionally.

3. References to USING phrase *parameters* operate according to the data description entries in the program's Linkage Section independent of their descriptions in the calling program.

4. A Linkage Section data item can be referred to in the Procedure Division only if it satisfies one of the following conditions:

   a. It is named in the Procedure Division's USING phrase.

   b. The address of the data item is assigned to another data item or a pointer. This is accomplished through the SET ADDRESS OF statement. See the Linkage Section in **section 5.3** and the **SET Statement** for additional information.

   c. It is subordinate to a *parameter* named in the USING phrase.

   d. It is a data item defined by a REDEFINES or RENAMES clause whose object is named in the USING phrase. Data items subordinate to such items may also be referenced.

   e. It is a condition-name or index-name associated with any data item that satisfies any of the previous conditions.

5. Any changes made to the data items referred to in the USING phrase are reflected in the calling program when the current program exits.

6. The CHAINING phrase is used only for a program that is the first program executed in a run unit (a "main" program). Each *parameter* has its initial value set according to the following rules:

   a. If the program is initiated from the host operating system, each *parameter* is initialized to the corresponding command-line argument.

   b. If the program is initiated by a CHAIN statement, then each *parameter* receives the value of the corresponding USING item specified by that CHAIN statement.

   c. Values are assigned to each *parameter* as if the value were the alphanumeric source for an elementary MOVE to *parameter*. If *parameter* is not alphanumeric, then it is treated as if it were implicitly redefined as alphanumeric before this MOVE occurs.

d. If there are fewer arguments than *parameters*, then the excess *parameters* are initialized according to the rules that would apply if they were not listed in the CHAINING phrase.

e. If there are more arguments than *parameters*, the excess arguments are ignored.

7. Segment-numbers describe the segments of the program. If segmentation is used, then sections must be placed so that a particular segment number is never smaller than the segment number for any previous section (in other words, all the sections belonging to a particular segment must be placed together).

8. Unless altered by the SEGMENT-LIMIT phrase of the Configuration Section, segment numbers less than 50 are placed in the program's fixed memory space. The fixed memory space is loaded when the program first starts and remains resident until the program is canceled.

9. Segment numbers of 50 or greater reside in the swapped memory space. The swapped memory space contains only one segment at a time. If the program's flow of control requires that a different segment be executed than the one that is currently residing in the swapped memory space, then it is loaded from the program's object file. This reduces the amount of memory space required by the program, at the expense of performing additional disk reads when new segments are needed. This swapped memory space applies only to Version 7.2 and earlier. Later versions treat segmentation as commentary.

10. Sections with the same *segment-no* are considered to be part of the same segment. They are all loaded together when that segment is read into memory.

11. The functional behavior of a program with segmentation is identical to that same program without segmentation. The only difference is that the program requires less memory but (potentially) more time to run. It is legal for the flow of control to pass directly or indirectly from one swapped segment to another.

12. When using segmentation, take care that the program's flow of control does not cause excessive swapping. If it does, the performance of the program may be substantially degraded.

13. An individual segment may not exceed 64 KB.

6.6 Procedure Division Statements

Every statement in COBOL is started with a reserved word called a verb. The following sections cover the verbs available in ACUCOBOL-GT.

ACCEPT Statement

The ACCEPT statement makes low-volume data available to the program. See **section 6.4.9, "Common Screen Options,"** for more information on many of the optional clauses shown below.

The different forms of the ACCEPT statement perform the following functions:

- **Format 1, ACCEPT** (an individual field)
- **Format 2, ACCEPT** (a Screen Section item, one or more fields)
- **Format 3, ACCEPT FROM** (returns selected data from the operating environment)
- **Format 4, ACCEPT SCREEN** (accepts data found on screen)
- **Format 5, ACCEPT ENVIRONMENT** (accepts value of variable from user's environment or from ACUCOBOL-GT's runtime configuration settings)
- **Format 6, ACCEPT** (ANSI style)
- **Format 7, ACCEPT** (activates a specific control)
- **Format 8, ACCEPT** (accepts HTML forms)
- **Format 9, ACCEPT EVENT** (accepts an "event")
- **Format 10, ACCEPT** (HP COBOL)
- **Format 11, ACCEPT** (HP COBOL)
- **Format 12, ACCEPT** (HP COBOL)

- **Format 13, ACCEPT FROM ENVIRONMENT-VALUE** (returns the value of the special register ENVIRONMENT-NAME)

Depending on the Format and options used, an ACCEPT statement can return the following predefined exception values. The numbers listed here are the actual values that are returned. Exception values can be found in the file "acugui.def". The names given here are the level 78 data items found in that file.

95 W-MESSAGE	- message received (ACCEPT ALLOWING MESSAGES)
96 W-EVENT	- ACCEPT terminated by an event
97 W-NO-FIELDS	- no input fields, or all fields protected or hidden
98 W-CONVERSION-ERROR	- numeric conversion error
99 W-TIMEOUT	- ACCEPT timed out (ACCEPT BEFORE TIME)

Please note that ACCEPT works differently on Windows and character-based systems. In Windows, an entry field is automatically in insert mode—that is, typing characters moves other characters instead of overwriting them. In character-based systems, entry fields start in overwrite mode, so typing new characters overwrites characters that already exist.

General Format

Format 1

```
ACCEPT {dest-item}
       {OMITTED  }
```

Remaining phrases are optional, can appear in any order.

```
AT screen-loc

{FROM} LINE NUMBER line-num
{AT  }

{FROM} {COLUMN  } NUMBER col-num
{AT  } {COL     }
```

```
       {POSITION}
       {POS     }

WITH PROTECTED SIZE length

WITH NO ADVANCING

{ERASE} [TO END OF] {LINE  }  (VAX, ICOBOL)
{BLANK}             {SCREEN}

{ERASE} [EOS]                 (RM)
{BLANK} [EOL]

WITH [NO] {BELL}
          {BEEP}

{UNDERLINE }
{UNDERLINED}

WITH {BLINKING}
     {BLINK   }

{HIGHLIGHT}
{HIGH     }
{BOLD     }
{LOWLIGHT }
{LOW      }
{STANDARD }

{REVERSE-VIDEO}
{REVERSE      }
{REVERSED     }

SAME

WITH {COLOR } color-val
     {COLOUR}

{FOREGROUND-COLOR } IS fg-color
{FOREGROUND-COLOUR}

{BACKGROUND-COLOR } IS bg-color
{BACKGROUND-COLOUR}

SCROLL [UP  ] [ BY scrl-num {LINE } ]
```

```
       [DOWN]               {LINES}

OUTPUT {JUSTIFIED}  {LEFT    }
       {JUST     }  {RIGHT   }
                    {CENTERED}
WITH {CONVERSION}
     {CONVERT   }

{WITH NO ECHO}
{NO-ECHO     }
{SECURE      }
{OFF         }

PROMPT [ CHARACTER IS prompt-lit ]

{DEFAULT [IS default]}
{UPDATE              }

ECHO

{AUTO         }
{AUTO-SKIP    }
{AUTOTERMINATE}
{TAB          }

{UPPER}
{LOWER}

CURSOR curs-offset

CONTROL cntrl-string

{REQUIRED   }
{EMPTY-CHECK}

{FULL        }
{LENGTH-CHECK}

{ZERO-FILL   }
{NUMERIC-FILL}

CONTROL KEY IN key-dest

BEFORE TIME timeout
```

```
ALLOWING MESSAGES FROM { THREAD thread-1 }
                       { LAST THREAD     }
                       { ANY THREAD      }

[ { ON EXCEPTION [key-dest] } {statement-1  } ]
  { ON ESCAPE    [key-dest] } {NEXT SENTENCE}
  { AT END                  }

[ NOT   { ON EXCEPTION } statement-2 ]
        { ON ESCAPE    }
        { AT END       }

[ END-ACCEPT ]
```

Format 2

```
ACCEPT screen-name
```

Remaining phrases are optional, can appear in any order.

```
AT screen-loc

{FROM} LINE NUMBER line-num
{AT  }

{FROM} {COLUMN  } NUMBER col-num
{AT  } {COL     }
       {POSITION}
       {POS     }

BEFORE TIME timeout

ALLOWING MESSAGES FROM { THREAD thread-1 }
                       { LAST THREAD     }
                       { ANY THREAD      }

[ UNTIL condition-1 ]

[ ON {EXCEPTION} statement-1 ]
     {ESCAPE   }

[ NOT ON {EXCEPTION} statement-2 ]
         {ESCAPE   }

[ END-ACCEPT ]
```

Format 3

```
ACCEPT dest-item FROM {DATE                          }
                      {DAY                           }
                      {CENTURY-DATE                  }
                      {DATE YYYYMMDD                 }
                      {CENTURY-DAY                   }
                      {DAY YYYYDDD                   }
                      {TIME                          }
                      {DAY-OF-WEEK                   }
                      {TERMINAL-INFO                 }
                      {SYSTEM-INFO                   }
                      {INPUT STATUS                  }
                      {ESCAPE KEY                    }
                      {LINE NUMBER                   }
                      {COMMAND-LINE                  }
                      {STANDARD OBJECT object-name}
                      {THREAD HANDLE                 }
                      {WINDOW HANDLE                 }
```

Format 4

```
ACCEPT dest-item FROM SCREEN
```

Remaining phrases are optional, can appear in any order.

```
AT screen-loc

{FROM} LINE NUMBER line-num
{AT  }

{FROM} {COLUMN  } NUMBER col-num
{AT  } {COL     }
       {POSITION}

SIZE length
```

Format 5

```
ACCEPT dest-item FROM {CONFIGURATION} env-name
                      {ENVIRONMENT  }

    [ ON EXCEPTION statement-1 ]

    [ NOT ON EXCEPTION statement-2 ]
```

```
    [ END-ACCEPT ]
```

Format 6

```
ACCEPT dest-item [ FROM mnemonic-name ]
```

Format 7

```
ACCEPT {control-handle}
       {CONTROL       }
```

Remaining phrases are optional, can appear in any order.

```
VALUE IN [ MULTIPLE ] value

AT screen-loc   [CELL  ]
                [CELLS ]
                [PIXEL ]
                [PIXELS]

AT LINE NUMBER line-num   [CELL  ]
                          [CELLS ]
                          [PIXEL ]
                          [PIXELS]

AT {COLUMN  } NUMBER col-num   [CELL  ]
   {COL     }                  [CELLS ]
   {POSITION}                  [PIXEL ]
   {POS     }                  [PIXELS]

AT CLINE NUMBER cline-num   [CELL ]
                            [CELLS]

AT CCOL NUMBER ccol-num   [CELL ]
                          [CELLS]

WITH {BELL}
     {BEEP}

BEFORE TIME timeout

CONTROL KEY IN key-dest

ALLOWING MESSAGES FROM { THREAD thread-1 }
                       { LAST THREAD     }
                       { ANY THREAD      }
```

```
ON  {EXCEPTION} [key-dest] statement-1
    {ESCAPE   }

NOT ON {EXCEPTION} statement-2
       {ESCAPE   }

END-ACCEPT
```

Format 8

```
ACCEPT external-form-item
```

Format 9

```
ACCEPT EVENT
```

Remaining phrases are optional.

```
BEFORE TIME timeout
ALLOWING MESSAGES FROM { THREAD thread-1 }
                       { LAST THREAD     }
                       { ANY THREAD      }

[ { ON EXCEPTION [code-dest] } {statement-1  } ]
  { ON ESCAPE    [code-dest] } {NEXT SENTENCE}
  { AT END                   }

[ NOT { ON EXCEPTION } statement-2 ]
      { ON ESCAPE    }
      { AT END       }

[ END-ACCEPT ]
```

Format 10 (HP COBOL)

```
ACCEPT identifier [FREE]  [FROM { SYSIN         }]
                                { CONSOLE       }
                                { mnemonic-name }
```

Format 11 (HP COBOL)

```
ACCEPT identifier FREE  [FROM { SYSIN         }]
                              { CONSOLE       }
                              { mnemonic-name }

[ON INPUT ERROR imperative-statement-1]
```

```
[NOT ON INPUT ERROR imperative-statement-2]

[END-ACCEPT]
```

Format 12 (HP COBOL)

```
ACCEPT identifier FROM { DATE        }
                       { DAY         }
                       { DAY-OF-WEEK }
                       { TIME        }
```

Format 13

```
ACCEPT value FROM ENVIRONMENT-VALUE
```

Syntax Rules

Note: For Syntax Rules and General Rules specific to Formats 10, 11, and 12, see Chapter 5, "HP COBOL Conversions," in *Transitioning to ACUCOBOL-GT.*

---

1. *Dest-item* is a data item that receives the accepted data. It must be USAGE DISPLAY unless the WITH CONVERSION phrase is also used.

   If the Format 3 STANDARD OBJECT, THREAD HANDLE, or WINDOW HANDLE phrase is used, *dest-item* must be a USAGE HANDLE data item of the appropriate type. If the WINDOW HANDLE phrase is used, *dest-item* can also be a PIC X(10) item.

2. *Screen-name* is the name of a screen entry declared in the program's Screen Section. See **section 5.8, "Screen Section,"** for more information.

3. *Screen-loc* is a numeric literal or data item containing either 4 or 6 digits. It may also be a group item of 4 or 6 characters. If a numeric item is used, it must be a non-negative integer.

4. In Format 7, the AT, LINE, COLUMN, CLINE, and CCOL phrases may be used only if the CONTROL phrase is also used.

5. *Line-num*, *col-num*, *cline-num*, and *ccol-num* are numeric data items or literals. Note that they may be non-integer, unless the value is in pixels.

6. *Length*, *color-val*, *curs-offset*, *timeout*, and *scrl-num* are numeric literals or data items. They must specify non-negative integers. You may also specify the value of *line-num*, *col-num*, *length*, and *color-val* with an arithmetic expression.

7. *Fg-color* and *bg-color* are integer literals or numeric data items. They may be arithmetic expressions. They may not be subscripted or reference modified. See **section 6.4.9**, "FOREGROUND-COLOR and BACKGROUND-COLOR Phrases," for a more detailed discussion of color settings and values.

8. *Prompt-lit* is a single-character alphanumeric literal or the figurative constant SPACE, ZERO, or QUOTE.

9. *Default* is a literal or data item. It specifies the default entry value.

10. *Key-dest* is a numeric data item. It receives the value of the key that terminated input.

11. *Thread-1* is a USAGE HANDLE or HANDLE OF THREAD data item.

12. *Statement-1* and *statement-2* are any imperative statements.

13. *Condition-1* is any conditional expression.

14. *Cntrl-string* and *env-name* are nonnumeric literals or data items.

15. *Mnemonic-name* must be a user-defined word declared in Special-Names that refers to a display device, or it must be the name of the display device itself. See **section 4.2.3, "Special-Names Paragraph,"** for a list of valid devices.

16. *Object-name* must be an alphanumeric literal or data item.

17. *Control-handle* must be USAGE HANDLE. If *control-handle* is a typed handle, it must be associated with a control. It should hold a handle returned by a DISPLAY Control-Type (Format 14) statement.

18. *Value* can be any data item. It receives the current value of the control when the ACCEPT statement terminates.

19. *Code-dest* is a numeric data item.
20. If the AT phrase is specified, neither the LINE nor the COLUMN phrase may be specified.
21. If the COLOR phrase is specified, neither the FOREGROUND-COLOR nor the BACKGROUND-COLOR phrase may be specified.
22. The CURSOR phrase may not be specified if a CURSOR phrase is specified in the program's Configuration Section.
23. AUTO, AUTO-SKIP, and AUTOTERMINATE are equivalent.
24. NO-ECHO, NO ECHO, OFF, and SECURE are equivalent.
25. BLANK and ERASE are equivalent.
26. BOLD, HIGH, and HIGHLIGHT are equivalent.
27. LOWLIGHT and LOW are equivalent.
28. COLUMN, COL, POSITION, and POS are equivalent.
29. BELL and BEEP are equivalent.
30. REVERSE-VIDEO, REVERSE, and REVERSED are equivalent.
31. BLINK and BLINKING are equivalent.
32. UNDERLINE and UNDERLINED are equivalent.
33. CONVERSION and CONVERT are equivalent.
34. REQUIRED and EMPTY-CHECK are equivalent.
35. FULL and LENGTH-CHECK are equivalent.
36. EXCEPTION and ESCAPE are equivalent.
37. If the OMITTED option is used, then none of the following phrases may be specified: SIZE, JUSTIFIED, CONVERT, NO ECHO, PROMPT, DEFAULT, ECHO, UPPER, LOWER, REQUIRED, FULL, ZERO-FILL, NUMERIC-FILL or any of the attribute setting phrases (such as COLOR or REVERSE).

38. The ERASE phrase has two forms. In VAX COBOL and ICOBOL compatibility modes, the ERASE LINE/SCREEN form must be used. In RM/COBOL mode, the ERASE EOS/EOL form must be used. This is indicated in the General Format by the symbols "*(VAX, ICOBOL)*" and "*(RM)*".

39. A NOT AT END phrase may not be paired with an ON EXCEPTION phrase, nor may a NOT ON EXCEPTION phrase be paired with an AT END phrase.

40. In a Format 6 ACCEPT statement, if you omit the FROM phrase, you must use the "-Ca" compile-time option. Otherwise, the compiler treats the statement as a Format 1 ACCEPT instead.

41. If the CONTROL KEY phrase is used, the *key-dest* option of the ON EXCEPTION phrase may not be specified.

42. You may add FROM CRT to a Format 1 ACCEPT statement to distinguish it from a Format 6 ACCEPT statement. You need to do this only if you use the "-Ca" compile-time option.

43. *External-form-item* is an input record for an HTML form when used in a Common Gateway Interface (CGI) program. It is a group data item (declared with the IS EXTERNAL-FORM clause) that has one or more elementary items associated with CGI variables. The association is made with the IS IDENTIFIED BY clause in the description of the elementary item(s).

    External-form-item may also be an output record for an HTML form. In this case, the group item is declared with both the IS EXTERNAL-FORM and IDENTIFIED BY clauses.

    See the IS EXTERNAL-FORM clause for more information about declaring external forms.

44. For Format 13, *value* is a Working-Storage data item of the type needed to receive the value of the environment or configuration value stored in ENVIRONMENT-NAME (see DISPLAY Format 17).

General Rules

Format 1 (ACCEPT dest-item)

1. The ACCEPT statement accepts data typed by the user. The data accepted is placed in *dest-item*. If the OMITTED option is used instead of *dest-item*, then the user is required to enter a termination key, and no data is transmitted to the program. This can be used if you want to enter a function key only and do not need any other data from the user.

2. The precise action of the ACCEPT statement depends both on the various clauses specified and the *compatibility mode* that the compiler is using. The compiler is run in RM/COBOL, VAX COBOL, or ICOBOL compatibility mode.

3. Use of the AUTO phrase causes a field to terminate as soon as it is filled with data--this is the default mode in RM COBOL. Use of the TAB phrase forces the user to finish with a termination key--this is the standard mode except for RM COBOL.

   The effects of the **CURSOR**, **CONTROL KEY**, **BEFORE TIME**, ALLOWING MESSAGES, **ON EXCEPTION**, and **AT END** clauses are described below. The effects of the remaining optional clauses are described in **section 6.4.9, "Common Screen Options."**

CURSOR Phrase

1. The CURSOR phrase specifies the initial cursor offset from the beginning of the field. The leftmost position of the ACCEPT field is offset one. If the CURSOR phrase is omitted or zero, then an offset of one is used.

2. The offset specified is reduced as needed to keep the cursor in the bounds of the field. Also, the offset is reduced to keep the cursor within the data contained in the field. This means that an offset of one will always be used if the DEFAULT (or UPDATE) phrase is not used.

3. If *curs-offset* is a numeric data item, then the ending cursor offset will be assigned to it when the ACCEPT statement terminates. This can be used to find the current cursor location (field column + *curs-offset* - 1).

4. If a CURSOR clause is specified in the program's Configuration Section, then that clause is used to determine the cursor's initial offset. See **section 4.2.3, "Special-Names Paragraph,"** for details.

CONTROL KEY Phrase

1. The CONTROL KEY clause allows for the entry of special keys and the recording of which key terminated input. (For additional information, see *ACUCOBOL-GT User's Guide*, section 4.3, "The Keyboard Interface.") When the CONTROL KEY clause is specified, all of the special keys named in the following table can be used.

2. *Key-dest* is a numeric item that receives the value of the key that terminated input. The keys allowed by the CONTROL KEY phrase and their returned values are:

Key	Value	Key	Value
Enter/Return	13	F-13	13*
Tab	9	F-14	14*
F-1	1*	F-15	15*
F-2	2*	F-16	16*
F-3	3*	F-17	17*
F-4	4*	F-18	18*
F-5	5*	F-19	19*
F-6	6*	F-20	20*
F-7	7*	Do (Command)	40*
F-8	8*	Up Arrow	52*
F-9	9*	Down Arrow	53*
F-10	10*	Page Up	67*
F-11	11*	Page Down	68*
F-12	12*	Help	90*

Control keys may also be used. They return their underlying ASCII value (for example, Control-B returns the value 2). If a field is entered in AUTO mode and the ACCEPT terminates because the field was filled

with data, then *key-dest* receives the value zero. If the BEFORE TIME phrase is specified and a time-out occurs, then *key-dest* receives the value "99".

3. The keys marked with "*" are exception keys. These keys may be used only if a CONTROL KEY or ON EXCEPTION phrase is specified for the ACCEPT statement (alternately, the "-Vx" compile-time option can be used). If these keys are not allowed, then the runtime system will ignore them when they are typed. Note that if you use the "-Ve" compile-time option to disable the ON EXCEPTION phrase for exception keys, then using the ON EXCEPTION phrase will not enable exception keys. (See also rule number 2 under ALLOWING MESSAGES Phrase.)

4. By using the "-Ve" compile-time option, you can cause ACUCOBOL-GT to return numeric conversion errors to the program. If you use this option to enable program handling of conversion errors, then *key-dest* will be assigned the value "98" whenever a conversion error occurs. See *ACUCOBOL-GT User's Guide*, section 4.3.2.1, "The Keyboard Variable," subsection 3, "CHECK-NUMBERS."

5. For compatibility with programs written under other systems, the values returned by these keys may be changed at runtime. Also, other keys may be added or removed from this table. See section 4.3.2, "Redefining the Keyboard," in Book 1, *ACUCOBOL-GT User's Guide*, for details on how to do this.

BEFORE TIME Phrase

1. The BEFORE TIME phrase allows you to automatically terminate an ACCEPT statement after a certain amount of time has passed. The *timeout* value specifies the time to wait in hundredths of a second. For example, "BEFORE TIME 500" specifies a timer value of 5 seconds.

2. The user must enter data to the ACCEPT statement before the timer elapses. As soon as the user starts entering data, the timer is canceled and the user may take as much time as desired to complete the entry. If the user does not enter any data before the timer elapses, then the ACCEPT statement terminates and returns a value exactly as if the user had typed the "enter" key. An exception condition is then raised and the exception key value is set to "99".

3. The precise amount of time waited can vary significantly from machine to machine. Factors involved include the rate of the system clock and the current load on the machine. For example, some UNIX systems update the system clock only once every second. The ACUCOBOL-GT runtime system takes the specified value and rounds it up to a whole number of "ticks" of the system clock. It then calls the system's alarm mechanism to interrupt the ACCEPT after this many "ticks" have elapsed. Different operating systems will handle this interrupt with varying degrees of accuracy. You should think of the *timeout* value as only an approximate one.

Note: If you want to program a task that executes at a regular interval, such as updating an on-screen clock, it is best to program the task in a separate thread (as opposed to timing out all of your ACCEPT statements). The periodic task would be placed inside a loop in a separate thread. The task would sleep for some period, wake and perform the update, and then loop back to the sleep state. This approach allows the runtime's thread manager to schedule the update, and ensures that no input, such as a mouse click, is missed when the update takes place. For more about threads, see Book 1, *ACUCOBOL-GT User's Guide*, section 6.8.

ALLOWING MESSAGES Phrase

1. The ALLOWING MESSAGES phrase causes the ACCEPT statement to terminate when a message is sent from the appropriate thread, as follows:

   a. The THREAD *thread-1* option allows messages from the thread identified by *thread-1*.

   b. The LAST THREAD option allows for messages from the "last" thread (see section 6.8.1, Book 1, *ACUCOBOL-GT User's Guide* for a discussion of the last thread).

   c. The ANY THREAD option allows messages from any thread.

2. If an allowed message is available when the ACCEPT begins, or one arrives while the ACCEPT is active, the ACCEPT terminates with an exception value of "95". The exception occurs even if exceptions are not otherwise allowed in the ACCEPT.

3. When an ACCEPT terminates due to a message, the target data item is not updated with the data entered by the user up to that point. This affects the following cases:

   a. In Format 1, *dest-item* is not updated.

   b. In Format 2, any VALUE, USING, or TO data items in the Screen Section are not updated.

   c. In Format 7, *value* is not updated.

Note: Testing of the data item is not performed (such as the REQUIRED phrase). Essentially, you get the termination status, but no other information. This allows you to share a data item between two controls. When the user changes one of them, it can send a message to the other to have that control update its appearance. If this rule was not in place, the message would terminate any ACCEPT on that control, and that control's current contents would overwrite the correct data in the shared data item. If you need to determine the current value of a control in response to a message, you can INQUIRE the control's value directly.

---

4. An ACCEPT statement that is suspended (because another thread has an active ACCEPT) terminates when an allowed message arrives.

5. If the ALLOWING MESSAGES phrase is omitted, messages will not terminate the ACCEPT. Instead, they are queued as normal.

ON EXCEPTION Phrase

Note: The "-Ve" compiler option alters the rules that determine which conditions cause an ON EXCEPTION phrase to receive control in a Format 1 ACCEPT statement. See **Section 2.2.11, "Video Options"** of the *ACUCOBOL-GT User's Guide* for details.

---

1. When this phrase is used, *statement-1* is executed when an exception condition occurs. An exception condition occurs under the following circumstances:

   a. An end-of-file condition occurs on the console.

b. A BEFORE TIME phrase was specified and the ACCEPT statement timed out.

c. An exception key was used to terminate input. This may be disabled with the "-Ve" compile-time option.

d. A conversion error occurred when numeric data was entered with the CONVERSION phrase. You must use the "-Ve" compile-time option to enable program handling of conversion errors for this exception condition to be used. Normally, the ACUCOBOL-GT runtime system does not allow invalid numeric data to be entered.

e. A *message* terminates the ACCEPT.

f. An *event* terminates the ACCEPT.

g. the screen contains no input fields, or all input fields are protected or disabled.

2. *Key-dest*, if specified, causes this phrase to have all of the effects of the CONTROL KEY phrase in addition to its normal effects. See the CONTROL KEY phrase above for details. If you specify *key-dest*, then exception keys will cause *statement-1* to execute even if you have used the "-Ve" compile-time option to turn off handling of exception keys by the ON EXCEPTION phrase.

3. If you specify an ON EXCEPTION phrase, then exception keys will be allowed for the ACCEPT statement. Otherwise exception keys will be disabled unless you use the CONTROL KEY phrase or the "-Vx" compile-time option. However, if you use the "-Ve" compile-time option to disable exception key handling, then an ON EXCEPTION phrase will *not* enable exception keys unless the *key-dest* option is also used.

4. If the NEXT SENTENCE option is used, then control will pass to the next executable sentence when an exception condition exists. Note that the ANSI standard states that "NEXT SENTENCE is an archaic feature and its use should be avoided."

5. If the NOT ON EXCEPTION phrase is specified, then *statement-2* executes if no exception condition exists.

6. The ON EXCEPTION phrase may not be specified if the AT END phrase is used. The *key-dest* option may not be used if the CONTROL KEY phrase is specified.

AT END Phrase

1. The AT END phrase causes the following *statement-1* to execute if an end-of-file occurs during the ACCEPT.

2. If NEXT SENTENCE is specified instead of *statement-1*, control passes to the next executable sentence if an end-of-file occurs. Note that the ANSI standard states that "NEXT SENTENCE is an archaic feature and its use should be avoided."

3. If the NOT AT END phrase is used, then *statement-2* executes if an end-of-file does not occur during the ACCEPT.

4. By default, ACUCOBOL-GT does not have a method of generating end-of-file conditions from a terminal. The runtime system can be reconfigured, however, to generate an end-of-file for selected keys. For details, see section 4.3.2, "Redefining the Keyboard," in Book 1, *ACUCOBOL-GT User's Guide*.

5. The AT END phrase may not be specified when the ON EXCEPTION phrase is.

Format 2 (ACCEPT screen-name)

1. This is a form level ACCEPT. A Format 2 ACCEPT statement allows the user to enter data into all of the input and update fields (as well as every control that can be activated) contained in *screen-name*. *Screen-name* must be a screen item declared in the program's Screen Section. After the user finishes, each input and update field is moved to its corresponding data item. See **section 5.8, "Screen Section,"** for a complete description of a screen item.

2. The AT, LINE, and COLUMN phrases describe the starting location of the screen item. These phrases are described in detail in **section 6.4.9, "Common Screen Options."** If either the line or column number is missing or zero, then line or column number 1 is used.

3. The BEFORE TIME phrase works in the same manner as described above for a Format 1 ACCEPT. If multiple input fields are being entered, then the timer is set only for the *first* field. As soon as the user starts entering data, as much time as desired may be taken to enter the screen.

4. The ALLOWING MESSAGES phrase works in the same manner as described above for a Format 1 ACCEPT.

5. If an exception condition occurs, the ON EXCEPTION phrase causes control to be transferred to *statement-1*. (Exception codes are stored in the variable named in SPECIAL-NAMES with the CRT STATUS phrase.) See rule number 1 under ON EXCEPTION phrase (above) for a list of the conditions that cause an exception. Exception keys are listed under the heading "CONTROL KEY Phrase" above. If an exception does not occur, and the NOT EXCEPTION phrase is specified, then *statement-2* executes.

6. Specifying the EXCEPTION phrase allows the user to terminate input with an exception key. Otherwise, exception keys are disabled unless the program is compiled with the "-Vx" option.

Note: You can code Screen Section entries that reference embedded procedures or an event procedure for Format 2 ACCEPT statements. See **section 5.9.6, "PROCEDURE Clause,"** for more information.

You can also code a Special-Names entry that allows an embedded procedure to control an ACCEPT statement. See **section 4.2.3, "Special-Names Paragraph,"** for more information.

7. If *condition-1* is specified, then the ACCEPT statement executes repeatedly until *condition-1* evaluates "true". The effect is exactly the same as coding:

```
PERFORM, WITH TEST AFTER, UNTIL condition-1
         ACCEPT screen-name, accept options
END-PERFORM
```

This can be convenient when you are retrieving data from a graphical screen. For example, you could use a statement like the following to get data from a dialog box until the user presses the box's "OK" button:

```
ACCEPT screen-1, UNTIL ok-button-pressed
```

Format 3 (ACCEPT FROM)

1. The Format 3 ACCEPT statement causes information to be transferred to *dest-item* according to the rules for the MOVE statement.

2. The DATE option causes the current date to be moved to *dest-item*. The date is composed of the year of the century, the month of the year, and the day of the month. Each element occupies two digits. For example, December 25, 1998, would be expressed "981225". DATE is treated as if it were described by a PICTURE 9(6) clause.

3. The DAY option causes the current date to be moved to *dest-item*. The format of the date is the year of the century (2 digits) followed by the day of the year (3 digits). For example, December 25, 1998, is "98359". DAY acts as if it were described by a PICTURE 9(5) clause.

4. ACCEPT FROM CENTURY-DATE returns the current date in the format "YYYYMMDD" (year/month/day). ACCEPT FROM DATE YYYYMMDD is equivalent to ACCEPT FROM CENTURY-DATE. ACCEPT FROM CENTURY-DAY returns the current date in the format "YYYYDDD" (year/day-of-year). ACCEPT FROM DAY YYYYDDD is equivalent to ACCEPT FROM CENTURY-DAY. These are the same as ACCEPT FROM DATE and ACCEPT FROM DAY, except that the year field is 4 digits instead of 2 digits.

   The compiler option "-Zy" lets you treat ACCEPT FROM DATE as ACCEPT FROM CENTURY-DATE, and ACCEPT FROM DAY as ACCEPT FROM CENTURY-DAY. For details, see the *ACUCOBOL-GT User's Guide* section 2.1.13, "Miscellaneous Options."

5. The TIME option causes the current time of day to be moved to *dest-item*. The format of the time is the hour (24-hour clock), the minutes, the seconds and the hundredths of a second. Each element occupies two digits. For example, 2:41 PM would be expressed "14410000". TIME acts as if it were described by a PICTURE 9(8) clause.

   Generally speaking, older UNIX machines keep time only to the nearest second. Thus the hundredths portion of the value returned by ACCEPT FROM TIME is typically zero on UNIX machines. However, System V Release 4 and some other UNIX versions have an alternate method for

determining the time with greater precision. The runtime system will use this alternate method if it is available, and then the hundredths portion will be filled in.

6. The DAY-OF-WEEK option causes the current day of the week to be moved to *dest-item*. The format of this item is a single digit where 1 represents Monday, 2 represents Tuesday, up through 7 for Sunday. DAY-OF-WEEK acts as if it were described by a PICTURE 9 clause.

7. The TERMINAL-INFO option causes information about the user's terminal to be moved to *dest-item*. This information is returned in the following format, as contained in the TERMINAL-ABILITIES group item defined in "acucobol.def":

```
01  TERMINAL-ABILITIES.
    03  TERMINAL-NAME                PIC X(10).
    03  FILLER                       PIC X.
        88  HAS-REVERSE              VALUE "Y".
    03  FILLER                       PIC X.
        88  HAS-BLINK                VALUE "Y".
    03  FILLER                       PIC X.
        88  HAS-UNDERLINE            VALUE "Y".
    03  FILLER                       PIC X.
        88  HAS-DUAL-INTENSITY       VALUE "Y".
    03  FILLER                       PIC X.
        88  HAS-132-COLUMN-MODE      VALUE "Y".
    03  FILLER                       PIC X.
        88  HAS-COLOR                VALUE "Y".
    03  FILLER                       PIC X.
        88  HAS-LINE-DRAWING         VALUE "Y".
    03  NUMBER-OF-SCREEN-LINES       PIC 9(3).
    03  NUMBER-OF-SCREEN-COLUMNS     PIC 9(3).
    03  FILLER                       PIC X.
        88  HAS-LOCAL-PRINTER        VALUE "Y".
    03  FILLER                       PIC X.
        88  HAS-VISIBLE-ATTRIBUTES   VALUE "Y".
    03  FILLER                       PIC X.
        88  HAS-GRAPHICAL-INTERFACE  VALUE "Y".
    03  USABLE-SCREEN-HEIGHT         PIC X(2) COMP-X.
    03  USABLE-SCREEN-WIDTH          PIC X(2) COMP-X.
    03  PHYSICAL-SCREEN-HEIGHT       PIC X(2) COMP-X.
    03  PHYSICAL-SCREEN-WIDTH        PIC X(2) COMP-X.
    03  FILLER                       PIC X.
        88  IS-REMOTE                VALUE "Y".
    03  CLIENT-MACHINE-NAME          PIC X(64).
```

```
03  FILLER                          PIC X.
    88  ACU-NO-TERMINAL             VALUE "Y".
03  CLIENT-USER-ID                  PIC X(20).
```

a. The TERMINAL-NAME field contains a short descriptive name of the terminal type being used (e.g., “vt100”, “tvi925”, “pc-color”). For the Windows runtime and thin client, the TERMINAL-NAME field contains the string “Windows”. The number-of-lines and number-of-columns fields contain the number of lines and columns the screen contains in its current configuration. The remaining fields contain a “Y” if the user’s terminal has the named feature and an “N” if it does not.

b. The NUMBER-OF-SCREEN-LINES and NUMBER-OF-SCREEN-COLUMNS fields hold the number of whole lines and columns (respectively) in the current floating window. If no window has been created, these values are set to the size of the default application window.

c. The HAS-VISIBLE-ATTRIBUTES field is set to “Y” if setting a video attribute on the terminal causes a character to be taken up on the screen. Terminals that behave in this fashion have additional restrictions over the terminals that have “hidden” attributes. For additional information on these restrictions, see *ACUCOBOL-GT User’s Guide*, section 4.5, “Restricted Attribute Handling.”

d. The HAS-GRAPHICAL-INTERFACE item is set to TRUE (value “Y”) if the runtime is using a graphical user interface. Otherwise, it is FALSE (value “N”). You can generally assume that graphical systems can support non-integer row/column positions and window/control sizes.

e. Non-graphical (i.e., character-based) systems will truncate non-integer row and column positions, along with window and control sizes to the nearest integer that is smaller (e.g., “line 1.5” will be treated as “line 1”).

f. The USABLE-SCREEN-HEIGHT field holds the height of the usable portion of the user’s display device in “base units.” This value represents character cells for character-based systems or pixels for graphical systems. On graphical systems, this value can be larger than the area used by your program (it is the largest area that a program could conceivably use). Under Windows 98, this

value excludes the Taskbar if the user has set the “Always on top” Taskbar option. On character-based systems, this value is the size of the actual screen, or the current application window if you are running in a character-emulator such as the Windows console-mode (DOS box) or an X-term.

g. USABLE-SCREEN-WIDTH is the same as USABLE-SCREEN-HEIGHT except that it returns the width instead of the height.

h. PHYSICAL-SCREEN-HEIGHT is the same as USABLE-SCREEN-HEIGHT except that it includes the entire screen instead of just the usable portion of the screen. Under Windows 98, this is the actual screen resolution. Under character-based systems, this value is normally the same as USABLE-SCREEN-HEIGHT because most character-based systems do not provide a method for measuring the physical screen.

i. PHYSICAL-SCREEN-WIDTH is the same as PHYSICAL-SCREEN-HEIGHT except that it returns the screen’s width instead of the screen’s height.

j. IS-REMOTE is set to TRUE (value “Y”) if the program is running with either the Thin Client. When IS-REMOTE is true: CLIENT-MACHINE-NAME is set to the name of the client that is running either **acuthin** plus a hyphen (“-”) and the hex value of the client process ID. For example:

```
techxp-2ef1
```

CLIENT-USER-ID is set to the client-side login name of the current user. This is the name the user entered when logging in to the client that is running acuthin. If it is not set, acuthin looks for the environment variable “USERNAME”. If that is not set, then the literal “USER” is placed in the field.

k. ACU-NO-TERMINAL is set to TRUE if the runtime was started with the “-b” option (“-b” inhibits terminal initialization and implies that no terminal is attached to the process). In this scenario, terminal I/O is undefined and should be avoided. When ACU-NO-TERMINAL is TRUE, other fields in TERMINAL-ABILITIES return their default value.

On many UNIX systems, the runtime can use ACCEPT FROM TERMINAL-INFO to determine the initial size of its window when it is running under the X Window system, Motif, OpenLook, or Sunview. It uses this information to set the number of rows and columns available to the program, and to scroll the screen correctly. The size found overrides the size specified in the "a_termcap" entry for the terminal. If you need to, you can override these settings with the LINES and COLUMNS environment variables.

8. The SYSTEM-INFO option causes some general information about the runtime to be moved to *dest-item*. The information is returned in the following format (SYSTEM-INFORMATION is defined in "acucobol.def"):

```
01  SYSTEM-INFORMATION.
    03  OPERATING-SYSTEM            PIC X(10).
        88  OS-IS-MSDOS             VALUE "MS-DOS".
        88  OS-IS-OS2               VALUE "OS/2".
        88  OS-IS-VMS               VALUES "VMS",
                                           "VAX/VMS".
        88  OS-IS-UNIX               VALUES "Unix",
                                           "Unix-V",
                                           "Unix-4",
                                           "UNOS".
        88  OS-IS-AOS                VALUE "AOS/VS".
        88  OS-IS-WINDOWS            VALUE "WINDOWS".
        88  OS-IS-WIN-NT             VALUE "WIN/NT".
        88  OS-IS-WIN-FAMILY         VALUES "WINDOWS",
                                           "WIN/NT".
        88  OS-IS-AMOS               VALUE "AMOS".
        88  OS-IS-MPE                VALUE "MPE/iX".
        88  OS-IS-MPEIX              VALUE "MPE/iX".
    03  USER-ID                      PIC X(12).
    03  STATION-ID                   PIC X(12).
    03  FILLER                       PIC X.
        88  HAS-INDEXED-READ-PREVIOUS   VALUE "Y".
    03  FILLER                          PIC X.
        88  HAS-RELATIVE-READ-PREVIOUS  VALUE "Y".
    03  FILLER                          PIC X.
        88  CAN-TEST-INPUT-STATUS       VALUE "Y".
    03  FILLER                          PIC X.
        88  IS-MULTI-TASKING            VALUE "Y".
    03  RUNTIME-VERSION.
        88  VERSION-PRIOR-TO-2-2        VALUE SPACES.
```

```
    05  RUNTIME-MAJOR-VERSION       PIC 99.
    05  RUNTIME-MINOR-VERSION       PIC 99.
    05  RUNTIME-RELEASE             PIC 99.
03  FILLER                          PIC X.
    88  IS-PLUGIN                   VALUE "Y".
03  SERIAL-NUMBER                   PIC X(20).
03  FILLER                          PIC X.
    88  HAS-LARGE-FILE-SUPPORT      VALUE "Y".
```

a. The OPERATING-SYSTEM field returns information about the operating system the runtime is ported to. It may contain one of the following values:

Value	Host System
Unix	Any version of Unix
MPE/iX	Running on HP machines
VMS	VMS or OpenVMS
WINDOWS	Microsoft Windows 98
WIN/NT	Microsoft Windows NT 4.0, or Windows 2000

These values are intended to give the program fundamental information about the runtime.

Note: The sample copy library “acucobol.def” includes some useful level 88 descriptions of operating system labels. For example, “OS-IS-UNIX” covers all UNIX platforms. For a complete list, see “acucobol.def” in the “samples” directory of your ACUCOBOL-GT installation.

---

b. The USER-ID field is filled in with the login name of the current user. If it is not set, the runtime looks for the symbol “USERNAME”. If it is not set, the literal “USER” is placed in this field.

c. The STATION-ID field is filled in with the station name of the video terminal attached to the executing program. For UNIX systems, the initial “/dev/” is removed from the name first.

On Windows machines, the value of the symbol “STATION” will be returned if it is set in the host environment. If it is not set, then the literal “CON” will be placed in this field.

If the program is displaying on a thin client, STATION-ID is filled with “at<*pid*>” where *pid* is the process ID of the runtime running on the server.

d. The HAS-INDEXED-READ-PREVIOUS field is “Y” if the READ PREVIOUS and START LESS THAN verbs are available for indexed files on the host system. If these verbs are not available, this field is set to “N”. If more than one indexed file system is being used, then this field is set to “Y” only if *all* of the file systems support READ PREVIOUS.

e. The HAS-RELATIVE-READ-PREVIOUS field is “Y” if the READ PREVIOUS and START LESS THAN verbs are available for relative files on the host system. If these verbs are not available, this field is set to “N”.

f. The CAN-TEST-INPUT-STATUS field is “Y” if the ACCEPT FROM INPUT STATUS verb is available on the host system. If not, it is set to “N”. Most machines can use this verb, but a few cannot. On these machines, executing this verb will return a constant value that you can pre-select with a configuration file option. There is a runtime configuration variable called **SCRIPT_STATUS** that controls the behavior of ACCEPT FROM INPUT STATUS when the input is not attached to a terminal.

g. The IS-MULTI-TASKING field is “Y” if the host system can run multiple tasks at once and has record locking facilities installed. This can be used to determine whether or not multiple copies of the runtime system can be run with correct file handling.

h. RUNTIME-VERSION fields are filled in with numbers that identify the major version number, minor version number, and release number. For example, ACUCOBOL-GT Version 3.2.0 would return 03 as the major version, 02 as the minor version, and 00 as the release number.

i. The IS-PLUGIN field is “Y” if the runtime is the Web Runtime running within a Web browser. Otherwise, it is set to “N”. See the separate book titled *A Programmer’s Guide to the Internet* for more information about the Web Runtime.

j. The SERIAL-NUMBER field is filled with the serial number of the runtime.

k. The HAS-LARGE-FILE-SUPPORT field is “Y” if the UNIX port in question has extended support for large sequential and relative files. Otherwise, it is set to “N”.

9. The INPUT STATUS form of the ACCEPT statement returns a value that indicates whether or not there is data currently available from the standard input. Normally, when an ACCEPT statement executes, the program pauses until the user types in some data. The INPUT STATUS option can be used to test to see if the user has entered some data before executing an ACCEPT.

   This can be useful in a loop that constantly updates information while still allowing for user input.

   The INPUT STATUS option causes this information to be moved to *dest-item*. It is treated as if the source item were described as PICTURE 9(1). The value is 1 if input is currently available, 0 if not.

   If the input is being redirected, by default the value returned is zero. You can control this with the configuration variables **INPUT_STATUS_DEFAULT** and **SCRIPT_STATUS**.

   A small number of hardware platforms do not have the ability to test for pending input. On these machines, this verb returns a constant value. By default, this value is zero, but it may be set to another value via the configuration variable INPUT_STATUS_DEFAULT.

Note: Another method of waiting for input while performing another operation is to use threads. A separate thread is used to perform the regular ACCEPT (for example), while the original thread performs the ongoing operation. For more about threads, see **Section 6.8, “Multiple Execution Threads”** of Book 1, *ACUCOBOL-GT User’s Guide*.

When you use the INPUT STATUS phrase of the ACCEPT verb with controls, and especially with ActiveX controls, the results may be undefined. A much better way of handling detection of user input in programs using controls is to employ event procedures or a separate thread for that purpose.

10. The ESCAPE option of the ACCEPT statement returns the termination key value of the last Format 1 or Format 2 ACCEPT statement. These values are listed under the heading "CONTROL KEY Phrase" above. The value is treated as a COMP-1 data item that is moved to *dest-item* using the standard rules for a numeric move (exception: in ICOBOL compatibility mode, the value is treated as a PIC 99 data item).

11. The ACCEPT FROM LINE NUMBER option returns a 3-digit number corresponding to the console device that is controlling the executing program. Because most of the machines that run ACUCOBOL-GT do not use device numbers, but use alphanumeric names instead, ACUCOBOL-GT computes the device number by the following procedure. First the device name is converted to upper case and hyphens are converted to underscores (on UNIX systems, the initial "/dev/" is removed). This name is then searched for in the environment (including the configuration file). If it is found, then the value of the name is returned. This allows you to assign a customized value for each device. If the name is not found in the environment, then a number is formed from any digits found in the original device name (for example, "tty15" would return as "15"). If there are no digits in the device name, the value 0 is used.

    If the program is displaying on a thin client, ACCEPT FROM LINE NUMBER returns the last three digits of the process ID of the runtime running on the server.

12. The ACCEPT FROM COMMAND-LINE option causes the contents of the original command line to be moved to *dest-item*. Only those elements of the command line that appear after the program name are returned. No parsing is done; you must parse the command line into separate arguments yourself.

    On Windows machines, the command line is limited to 1024 characters. The command line remains unchanged regardless of the actions of any CHAIN, CALL PROGRAM, or CALL RUN verbs (which all start new run units).

For an alternate method in which the runtime system parses the arguments for you, see the **CHAIN Statement**.

To change the contents of the command-line buffer, see the DISPLAY UPON COMMAND-LINE statement (Format 8) in this section. You can also access the command-line buffer from a C program. The buffer is an external data array named Acmd_line.

13. The STANDARD OBJECT option returns a handle to one of the system's pre-defined resources. The resource returned depends on the value of *object-name*, as described in the following table:

Object-Name	**Resource**
FIXED-FONT	The host system's default fixed-size font. This is the default font used for textual displays. Under Microsoft Windows, this font is Windows' SYSTEM-FIXED-FONT.
TRADITIONAL-FONT	Also a fixed-size font. This font uses the standard character set associated with the host hardware (as opposed to the host graphical system). On many systems, this font is the same as FIXED-FONT. Under Microsoft Windows, this font is Windows' OEM-FIXED-FONT and uses the "OEM" character set instead of the "ANSI" character set.
DEFAULT-FONT	This is the default font used by controls. On a graphical system, this font is usually a proportional font, and usually the same as SMALL-FONT. On non-graphical systems, this is the same as the FIXED-FONT. You can specify the default font with the DEFAULT_FONT configuration variable (see **DEFAULT_FONT** in Appendix H, Book 4, *Appendices*).
LARGE-FONT	A moderately large font that is appropriate for controls. On graphical systems, this is a proportionally spaced font. Under Microsoft Windows, this is Windows' SYSTEM-FONT. On non-graphical systems, this is the same as FIXED-FONT. On some systems, this is the same as MEDIUM-FONT.

Object-Name	Resource
MEDIUM-FONT	An average size font that is appropriate for controls. On graphical systems, this is a proportionally spaced font. Under Microsoft Windows, this font is a boldface version of Windows' ANSI-VAR-FONT. On non-graphical systems, this is the same as FIXED-FONT.
SMALL-FONT	A small font that is appropriate for controls. On graphical systems, this is a proportionally spaced font. Under Microsoft Windows, this font is Windows' ANSI-VAR-FONT. On non-graphical systems, this is the same as FIXED-FONT. On some systems, this is the same as MEDIUM-FONT.
LM-RESIZE	A standard layout manager that assists in moving or resizing controls when a window changes size. See Book 2, section 4.8, "Layout Managers."

You may use either upper-case or lower-case values in *object-name*. If *object-name* does not match any of the allowed values, *dest-item* is set to NULL.

14. The ACCEPT FROM THREAD statement moves the thread ID of the executing thread to dest-item.

15. The WINDOW HANDLE option causes dest-item to receive the handle of the initial or current floating window. Note that this is the only way to get the handle of the default main application window. You do this by performing an ACCEPT FROM WINDOW HANDLE prior to creating any floating windows in your application.

Format 4 (ACCEPT SCREEN)

1. The ACCEPT FROM SCREEN verb returns data present on the user's terminal. This differs from the Format 1 ACCEPT statement in that the user does not enter the data. Instead, the data is what is already present on the screen. This can be used to obtain an image of the user's current screen. Note however that screen attributes such as underlines and reverse colors are not returned.

On graphical systems, this verb does not return any information contained in controls. To determine the current value of a control, use the INQUIRE verb.

2. *Dest-item* must specify an alphanumeric data item without the JUSTIFIED phrase. It is filled with the returned screen contents. *Dest-item* is space-filled on the right if it is larger than the returned screen contents.

3. The LINE phrase specifies the screen line to use for the ACCEPT. Line one refers to the top line of the current window. If the LINE phrase is missing or *line-num* is zero, the current cursor line is used.

4. The COLUMN phrase specifies the screen column to use. Column one refers to the leftmost column of the current window. If the COLUMN phrase is missing or zero, then the column depends on the following:

   a. If the LINE phrase is used (and is not zero), then column one is used.

   b. Otherwise, the current cursor column is used.

5. The SIZE phrase specifies the number of screen positions to return. If the SIZE phrase is missing, then the size of *dest-item* is used. If the SIZE phrase specifies fewer characters than the size of *dest-item*, then *dest-item* is space-filled on the right.

6. The ACCEPT FROM SCREEN verb does not change the cursor position or modify the screen in any way. A single ACCEPT FROM SCREEN can “wrap around” the right edge of the window to return characters from multiple screen lines.

7. Except as specified in the following rule, all characters on the terminal screen are returned using the underlying representation of the character in the native character set. This is usually the ASCII value of the character.

8. The ACUCOBOL-GT Terminal Manager returns consistent values for certain special characters, regardless of the hardware being used. This is done for certain characters used by the Terminal Manager that do not have an ASCII representation. These special characters are the following:

Character	**Value**
Unknown	1
Horizontal Line	2
Vertical Line	3
Upper Left Corner	4
Upper Right Corner	5
Lower Left Corner	6
Lower Right Corner	7
“Tee” Up	8
“Tee” Right	9
“Tee” Down	10
“Tee” Left	11
4-Way Intersection	12
Bottom Endpoint	13
Left Endpoint	14
Top Endpoint	15
Right Endpoint	16
Visible Attribute	17

The “Unknown” value is returned for any screen position that cannot be determined by ACUCOBOL-GT. The “Visible Attribute” character is used by certain types of terminals when they are displaying video attributes. On these types of terminals, video attributes take up screen positions. They usually appear as spaces on the screen. The remaining special characters are various line segments used by the DISPLAY WINDOW, DISPLAY LINE, and DISPLAY BOX verbs.

9. **Technical Note:** It is often desirable to translate the special case characters into ASCII values. You can do this easily with the INSPECT CONVERTING verb. For example, the following statement converts the line-drawing characters into hyphens, vertical bars, and plus signs, and translates the "unknown" and "visible attribute" characters into spaces:

```
INSPECT ... CONVERTING
        X'0102030405060708090A0B0C0D0E0F1011'
    TO " -|+++++++++|-|- "
```

Format 5 (ACCEPT ENVIRONMENT)

1. A Format 5 ACCEPT statement returns values from the user's environment or the ACUCOBOL-GT runtime system's configuration settings. *Env-name* is the name of the environment setting whose value is to be returned. If you provide the literal name of this item (such as CURSOR-MODE), you must enclose it in quotes. The value returned from this item is moved to *dest-item*.

2. This verb will search for *env-name* in the following places:

   a. First, the runtime system sees if *env-name* matches any of its configuration variables. If it does, the configuration variable's current setting is moved to *dest-item*. Not all configuration variables can be returned by the ACCEPT verb, because some of them have multiple settings. The list of configuration variables that can be returned is detailed in Appendix H, Book 4, *Appendices.*

   b. Next, *env-name* is searched for in the runtime system's local environment. This environment consists of all of the entries in the ACUCOBOL-GT configuration file plus any entries made with the SET ENVIRONMENT verb. This excludes the runtime system's configuration variables that are covered in rule (a) above. Note that any entry in the ACUCOBOL-GT configuration file that is also in the user's host environment has its *initial* value set the same as the entry in the user's environment.

   c. Finally, on machines that have a user-maintained environment, that environment is searched for *env-name*. A description of the user's environment for each machine is given in section 1.4 of the *ACUCOBOL-GT User's Guide*.

When the runtime system searches for *env-name*, only the first 30 characters are used. Also, any lower-case characters in *env-name* are treated as upper-case, and any hyphens are treated as underscores. If the truncated name is not found, the runtime searches again, this time looking for *env_name* exactly as specified.

3. If an entry is found, then its value is moved to *dest-item*. For numeric configuration variables, the source item is treated as if it were a COMP-1 data item. For all other entries, the source item is treated as an alphanumeric data item. The value is moved to *dest-item* according to the rules for the MOVE statement. Note that if the source item is numeric, then *dest-item* may be defined either as a numeric field or as an alphanumeric field of five or more characters. If *dest-item* is alphanumeric and is larger than five characters, the value that is returned will occupy the leftmost five characters of *dest-item*.

4. If no matching entry is found, or if the *env-name* is the name of a configuration variable whose value cannot be returned, spaces are moved to *dest-item* and *statement-1*, if specified, is executed.

5. If a legal matching entry is found, then *statement-2* (if specified) is executed.

Format 6 (ANSI ACCEPT)

1. A Format 6 ACCEPT statement reads a line of input from the standard input device (usually the user's console). This data is then moved into *dest-item*. If the data is longer than the size of *dest-item*, it is truncated on the right. If it is smaller, then it is space-filled on the right.

2. Prior to reading the line, the runtime system places the user's terminal into the state it normally occupies when it is used by the operating system. The runtime then requests the input from the operating system. The operating system usually provides some form of input editing, such as backspacing. The exact editing available depends on the host operating system.

3. **Technical Note:** Because this verb requests input directly from the operating system, ACUCOBOL-GT's Terminal Manager is not aware of the changes that are occurring to the screen. This can cause problems if you mix ANSI-style and ACUCOBOL-GT-style ACCEPT and DISPLAY verbs in the same program. On many machines, ACUCOBOL-GT's Terminal Manager maintains an image of the

user's screen. (This improves efficiency by removing redundant screen output and is also used to implement "pop-up" windows.) Bypassing the Terminal Manager can cause the Terminal Manager's screen image to become incorrect. This can cause strange effects when it is mixed with an ACUCOBOL-GT-style ACCEPT or DISPLAY verb, including:

- lost data
- incorrect functioning of CLOSE WINDOW
- incorrect cursor position
- incorrect character attributes
- incorrect display in debugger

For these reasons, you must be careful when using ANSI ACCEPT. Here are some useful guidelines:

a. If your statement does not affect the screen image, then it can be used safely. For example, sending a control sequence to an attached cash register is safe.

b. If you use only ANSI-style ACCEPT and DISPLAY verbs, then you should not experience any problems except that the debugger will not be able to show the user's screen.

c. If you must mix formats, then you can use the library routine "W$FORGET" to correct the behavior of the Terminal Manager. This routine causes the Terminal Manager to enter its initial state. It will assume that it does not know the screen image or current attribute settings. Calling this routine after a series of ANSI-style ACCEPT or DISPLAY verbs will place the Terminal Manager into a state where it can operate correctly. See Appendix I, Book 4, *Appendices* for additional information.

d. You can safely use the verb if your ANSI-style ACCEPT or DISPLAY sends data to a device other than the user's console, such as the standard error file.

Note: When you are running a 32-bit Windows CGI program with the "-f" option or with the A_CGI environment variable set, the runtime reads only the number of bytes specified by the web server in the CONTENT_LENGTH environment variable. The runtime does not wait for an end of file condition. If you are running without the "-f" option and have not set the A_CGI environment variable, then the runtime reads until an end of file condition occurs. Note that some web servers such as Microsoft Internet Information Server 4.0 do not terminate input to a CGI program with an end of file condition. Instead, they rely upon the CGI program to read exactly the number of bytes specified in the CONTENT_LENGTH variable. When running an ANSI ACCEPT style CGI program using these web servers, you must use the "-f" option or set the A_CGI environment variable.

Format 7 (ACCEPT control-handle)

1. The ACCEPT Control-Handle statement activates the control identified by *control-handle*. The user interacts with the control until some terminating event occurs. The event that caused the termination is then stored in *key-dest*, and the control's current value is stored in *value*. The ACCEPT statement then terminates.

2. If the CONTROL phrase is used, the runtime activates the control located at the screen position specified by the AT, LINE, and COLUMN phrases in the current window (on non-graphical systems, the CLINE and CCOL phrases also apply). The runtime maintains a list of controls in each window. When attempting to activate a control at a specific location, the runtime searches this list, using the first control it finds that exactly matches the given location. The list is maintained in order in which the controls are created. If the runtime does not find a control at the specified location, it returns an exception value of "96" (the same as doing an ACCEPT of a invalid control handle).

   The following example creates an anonymous entry field and then ACCEPTs it, using its screen position.

```
DISPLAY ENTRY-FIELD, LINE 2, COL 5, SIZE 15.
ACCEPT CONTROL, VALUE MY-DATA, LINE 2, COL 5.
```

3. *Key-dest* names a data item that will receive a code indicating the terminating event. The program's CRT STATUS (if any) also receives the termination code.

4. When the ACCEPT statement terminates, the current value of the control is moved to *value* in accordance with the rules for the MOVE statement. The type of control determines the source format of the value.

5. The BEFORE TIME phrase operates in the same manner as it does in a Format 1 ACCEPT statement.

6. The WITH BELL phrase causes the station's bell to sound when the control is initially activated.

7. The ALLOWING MESSAGES phrase works in the same manner as described above for a Format 1 ACCEPT.

8. The ACCEPT statement can terminate in several different ways. These ways are classified as either *normal* terminations or as *exceptions*. If an exception caused the termination, then *statement-1* is executed. Otherwise, *statement-2* is executed. If you do not specify the ON EXCEPTION phrase, then the ACCEPT statement ignores exception keys (function keys). You can override this behavior with the "-Vx" compile-time option.

   If you attempt to ACCEPT a disabled or otherwise invalid control, or if the active control is hidden during the ACCEPT either by an event procedure or by code running in a different thread, the ACCEPT will immediately terminate, returning a CRT STATUS value of "97". For a Screen Section ACCEPT, this will occur only if all of the referenced controls are disabled.

9. You may also activate a control with a Format 2 ACCEPT statement (a Screen Section ACCEPT) . If the referenced Screen Section entry defines any controls, they are activated as appropriate.

Format 8 (ACCEPT external-form-item)

1. The "external form" of Format 8 is called an "output form" if the IDENTIFIED BY clause is used to associate it with an HTML template file. If the IDENTIFIED BY clause is omitted, it is called an "input form".

For example, the following is an input form:

```
01  CGI-FORM IS EXTERNAL-FORM.
    03  CGI-VAR1  PIC X(10).
    03  CGI-VAR2  PIC X(10).
```

and this is an output form:

```
01  HTML-FORM IS EXTERNAL-FORM IDENTIFIED BY
    "tmplate1".
    03  HTML-VAR1  PIC X(10).
    03  HTML-VAR2  PIC X(10).
```

2. The ACCEPT verb treats input and output forms the same. ACCEPT sets the value of each elementary item, in order, to the value of its associated CGI variable, padding with trailing spaces. ACCEPT automatically decodes and translates the CGI input data before moving it to the elementary items of *external-form-item*. The value of each CGI variable is converted to the appropriate COBOL data type when it is moved to the external form.

   CGI variable names are case-sensitive. However, for convenience, if ACCEPT cannot identify a CGI variable, it will repeat the search for the variable ignoring the case.

3. If the CGI variable is empty or does not exist, ACCEPT sets the value of numeric data items to zero and nonnumeric data items to spaces. This behavior is configurable. If you do not want ACCEPT to clear the value of the data item when its CGI variable does not exist in the CGI input data, set the CGI_CLEAR_MISSING_VALUES configuration variable to “0” (off, false, no). See the entry for **CGI_CLEAR_MISSING_VALUES** in Appendix H for more details.

4. If the CGI variable is repeated in the CGI input data (as it would be in the case where multiple items have been selected from a “choose-many” list), the external form item that is identified with the CGI variable must be in a table. Otherwise, only the first CGI value is moved to the external form item.

   For example:

```
01  CGI-FORM IS EXTERNAL-FORM.
    03  CGI-TABLE    OCCURS 10 TIMES.
        05  CGI-VAR1   PIC X(10).
```

```
        05  CGI-VAR2   PIC X(10).
```

or

```
01  CGI-FORM     IS EXTERNAL-FORM.
    03  CGI-VAR1  PIC X(10) OCCURS 10 TIMES.
    03  CGI-VAR2  PIC X(10) OCCURS 10 TIMES.
```

ACCEPT moves the values of the CGI variable to the items in the table. After all of the CGI values have been moved to items in the COBOL table, the remaining items in the table are set to 0 if they are numeric items, and set to spaces otherwise.

Format 9 (ACCEPT EVENT)

1. The ACCEPT EVENT statement waits for a terminating event to occur. The event that caused the termination is stored in *code-dest*. The ACCEPT statement then terminates.

2. *Code-dest* names a data item that will receive a code indicating the terminating event. The program's CRT STATUS (if any) also receives the termination code.

3. The BEFORE TIME phrase operates in the same manner as it does in a Format 1 ACCEPT statement.

4. The ALLOWING MESSAGES phrase works in the same manner as described above for a Format 1 ACCEPT.

5. ACCEPT EVENT is similar to ACCEPT OMITTED except that it does not display a default initial window and it does not detect keyboard termination or exception keys.

6. The ACCEPT EVENT statement is designed for use in programs without a user interface. However, if the program has a user interface and interacting with it causes a terminating event to occur, ACCEPT EVENT will terminate. In the debugger, ACCEPT EVENT will terminate if the user presses the Enter key.

7. The ACCEPT EVENT statement can terminate in several different ways. These ways are classified as either normal terminations or as exceptions. If an exception caused a termination, then statement-1 is executed. Otherwise, statement-2 is executed.

Code Example

Format 9:

The following program uses the Microsoft Agent Control (ActiveX) to create a “genie” character and then directs it to speak “Hello World”. It does not have a visible initial window or a user interface that allows keyboard or mouse input.

```
identification division.
program-id.  no-ui.
environment division.
special-names.
copy "msagent.def".
                  .
data division.
working-storage section.

77  genie1-handle       usage handle of Agent.
77  request-handle      usage handle of IAgentCtlRequest.
77  request-status      pic 9.

procedure division.
Main-Logic.

   display initial window visible = 0.

   display Agent of AgentObjects handle in genie1-handle.

   modify genie1-handle
            Characters::Load("Genie1", "genie.acs").

   use genie1-handle Characters::Item("Genie1")

       modify ^LanguageID 1033
              ^Show()
              ^Speak "Hello World" giving request-handle

       perform until request-status = 1
           ACCEPT EVENT BEFORE TIME 1000
           inquire request-handle status in request-status
       end-perform

       destroy request-handle
```

```
        modify ^Hide()

    end-use.

    destroy genie1-handle.

    stop run.
```

Formats 10, 11, 12

See Chapter 4, "HP COBOL Conversions," of *Transitioning to ACUCOBOL-GT.*

Format 13 (ACCEPT FROM ENVIRONMENT-VALUE)

1. Use a Format 13 ACCEPT to fetch the value of an environment or configuration variable stored with a Format 17 DISPLAY statement (DISPLAY UPON ENVIRONMENT-NAME).
2. The *value* data item should be of a size and type that will accommodate the value of the environment or configuration variable.

ADD Statement

The ADD statement performs arithmetic addition.

General Format

Format 1

```
ADD {num} ... TO { result [ROUNDED] } ...

   [ ON SIZE ERROR statement-1 ]

   [ NOT ON SIZE ERROR statement-2 ]

   [ END-ADD ]
```

Format 2

```
ADD {num} ... TO num GIVING { result [ROUNDED] } ...

   [ ON SIZE ERROR statement-1 ]
```

```
   [ NOT ON SIZE ERROR statement-2 ]

   [ END-ADD ]
```

Format 3

```
ADD {CORRESPONDING} group-item TO group-item [ROUNDED]
    {CORR         }

   [ ON SIZE ERROR statement-1 ]

   [ NOT ON SIZE ERROR statement-2 ]

   [ END-ADD ]
```

Format 4

```
ADD TABLE src-table TO dest-table [ROUNDED]

   [ FROM INDEX src-start TO src-end ]

   [ DESTINATION INDEX dest-start ]

   [ ON SIZE ERROR statement-1 ]

   [ NOT ON SIZE ERROR statement-2 ]

   [ END-ADD ]
```

Syntax Rules

1. *Num* is a numeric literal or elementary numeric data item.
2. *Result* is an elementary numeric data item or, in Format 2, an elementary numeric edited data item.
3. *Group-item* is a group item containing one or more elementary numeric data items.
4. *Statement-1*, and *statement-2* are imperative statements.
5. CORR is an abbreviation of CORRESPONDING.

6. *Src-table* and *dest-table* are numeric data items that are table elements. The low-order subscript of these items must be omitted. For example, if "SRC-1" was an element of a one-dimensional table, then you would just use "SRC-1" in the statement. If "SRC-2" was an element of a two-dimensional table, and you wanted to add all the elements in row "2", you would use "SRC-2(2)".

7. *Src-start, src-end* and *dest-start* are numeric literals or data items. These items may not be subscripted.

General Rules

1. Note that pertinent additional information is located in the sections covering **Arithmetic Operations** (6.4.1), **Multiple Receiving Fields** (6.4.2), the **ROUNDED Option** (6.4.3), the **SIZE ERROR Option** (6.4.4), and the **CORRESPONDING Option** (6.4.5).

2. In Format 1, all *num*s are added together and their sum is then added to each *result* in turn.

3. In Format 2, all *num*s are added together and their sum is moved to each *result* field.

4. In Format 3, each pair of corresponding elementary numeric items in the two *group-items* are added together. The results are moved to the corresponding items in the *second group-item*.

5. In Format 4, a range of *src-table* elements is added to a range of *dest-table* elements. The results are stored in *dest-table*. The first element of the *src-table* range is added to the first element of the *dest-table* range, the second element to the second, and so on.

6. *Src-start* specifies the first element of the source range. If omitted, the value defaults to "1". *Src-end* specifies the last element of the range (inclusive). If omitted, it is set to the current upper bound of the source table. In a multidimensional table, the range of elements varies over the innermost OCCURS.

7. *Dest-start* indicates the first element of the destination range. If omitted, it defaults to "1". Note that the last element of the destination range is *dest-start + src-end -1*.

8. If the SIZE ERROR phrase is used, elements for which the size error condition occurs are not updated; other elements are updated. When an add results in a size error, *statement-1* executes, otherwise *statement-2* executes.

Note: A Format 4 ADD TABLE statement is usually substantially faster than an equivalent PERFORM loop. The degree of improvement depends on the size of the range (larger ranges show better improvement). The SIZE ERROR and ROUNDED phrases typically add significant overhead. The runtime always performs table boundary checking in ADD TABLE, even if the program is not compiled with "-Za".

Code Example

Format 4:

The following definitions will be used in the examples:

```
01  SOURCE-TABLE OCCURS 20 TIMES    PIC S9(9)V99.
01  DEST-TABLE OCCURS 20 TIMES      PIC S9(9)V99.

01  ROLL-UP-TABLE.
    03  TOTALS OCCURS 10 TIMES.
        05  REPORT-SUM
            OCCURS 20 TIMES         PIC S9(9)V99.
77  CTR                             PIC 99.
```

To add all the elements of SOURCE-TABLE to DEST-TABLE:

```
ADD TABLE SOURCE-TABLE TO DEST-TABLE
```

To add the first five elements of SOURCE-TABLE to the last five elements of DEST-TABLE:

```
ADD TABLE SOURCE-TABLE TO DEST-TABLE
     FROM INDEX 1 TO 5
     DESTINATION INDEX 16
```

To add all the REPORT-SUM elements in the last TOTALS row to the row "above" it (second to last row):

```
ADD TABLE REPORT-SUM(10) TO REPORT-SUM(9)
```

To perform the same operation using a PERFORM loop you would have to write the following code:

```
PERFORM VARYING CTR FROM 1 BY 1 UNTIL CTR > 20
    ADD REPORT-SUM(10, CTR) TO REPORT-SUM(9, CTR)
END-PERFORM
```

ALTER Statement

The ALTER statement changes the destination of a GO TO statement.

General Format

```
ALTER { goto-proc TO [ PROCEED TO ] new-proc } ...
```

Syntax Rules

1. *Goto-proc* is the name of a paragraph consisting of a single GO TO statement without the DEPENDING phrase.

2. *New-proc* is a procedure name.

General Rules

1. The ALTER statement changes the destination of the GO TO statement in *goto-proc*.

2. After execution of the ALTER statement, the GO TO in *goto-proc* causes control to transfer to *new-proc*.

3. The ALTER verb has been declared an obsolete element of COBOL by the COBOL standards committee. It is recommended that instances of ALTER be replaced by the GO TO verb using the DEPENDING ON phrase.

CALL Statement

The CALL statement causes control to be transferred to another program. For a discussion of how the runtime handles program calls, see section 2.9, "Calling Subprograms," in Book 1. For information about calling subroutines in DLLs and UNIX shared libraries, see Chapters 3 and 6 of *A Guide to Interoperating with ACUCOBOL-GT.*

General Format

Format 1

```
CALL [IN THREAD] program-name

 [ HANDLE IN handle-1 ]

 [ {RETURNING} INTO return-val ]
   {GIVING   }

 [ ON     {EXCEPTION} statement-1 ]
          {OVERFLOW }

 [ NOT ON {EXCEPTION} statement-2 ]
          {OVERFLOW }

 [ END-CALL ]
```

where *size-phrase* is:

```
[WITH MEMORY SIZE {= } memory-size]
                  {IS}
```

Format 2

```
CALL RUN program-name

 [ USING {parameter} ... ]

 [ ON     {EXCEPTION} statement-1 ]
          {OVERFLOW }

 [ NOT ON {EXCEPTION} statement-2 ]
          {OVERFLOW }
```

```
 [ END-CALL ]
```

Format 3

```
CALL PROGRAM program-name

 [ USING {parameter} ... ]

 [ ON  {EXCEPTION} statement-1 ]
       {OVERFLOW }

 [ END-CALL ]
```

Format 4 (HP COBOL)

```
CALL { identifier-1           } [ USING { \\            } ... ]
     { [ INTRINSIC ] literal-1 }         { @identifier-2  }
                                         { identifier-2   }
                                         { literal-2      }
                                         { \identifier-2\ }
                                         { \literal-2\    }

     [ GIVING identifier-3 ]

     [ ON {EXCEPTION} statement-1 ]
          {OVERFLOW }

     [ NOT ON {EXCEPTION} statement-2 ]
              {OVERFLOW }

     [ END-CALL ] ]
```

Syntax Rules

Note: For Syntax Rules and General Rules specific to Format 4, see Chapter 4, "HP COBOL Conversions," in *Transitioning to ACUCOBOL-GT.*

1. *Program-name* is a nonnumeric literal or an alphanumeric data item.
2. *Handle-1* must be a USAGE HANDLE or HANDLE OF THREAD data item.
3. The NULL and OMITTED options are synonymous.

4. *Memory-size* is a numeric literal or data item.

5. *Parameter* is any non-level 88 data item or a literal. It may be subscripted and reference modified. It may be the SELECT name of an open COBOL file.

6. *Statement-1* and *statement-2* are imperative statements.

7. EXCEPTION and OVERFLOW are equivalent.

8. The NOT EXCEPTION phrase may not be used when the PROGRAM option is used.

9.

10. RETURNING and GIVING are synonymous.

General Rules

1. *Program-name* specifies the name of the called program. The exact procedure for resolving this name into the name of a program to call is described in Book 1, *ACUCOBOL-GT User's Guide*, section 2.9.

2. The THREAD option starts a new thread before calling the target program. The called program executes in the new thread. The calling program continues to execute in parallel. When the called program exits, its thread is terminated.

3. The HANDLE phrase stores the ID of the thread in *handle-1* immediately after the new thread starts. This occurs before the USING parameters are evaluated, so that it is possible to pass *handle-1* to the called program.

4. The *parameters*' order of appearance in the USING phrases of the CALL verb, and the called program's Procedure Division header, determine the correspondence between the data names used in the called and calling programs. This correspondence is established by position, not by name.

5. The SIZE phrase is used when the parameter in the USING phrase is a memory address (pointer to memory) and you need to specify the size of the piece of memory that is located at that address. The SIZE phrase

supports the calling of DLLs on the display host by thin client applications. For complete information, see section 7.2.6, "Calling Dynamic Link Libraries (DLLs)," in the *AcuConnect User's Guide*.

6. The BY phrase controls how parameters are passed to the called program. The default method is BY REFERENCE. This method causes the address of the data item to be passed to the receiving program. This is the only method of passing data available in versions of ACUCOBOL-GT prior to 2.0.

   The BY CONTENT phrase is similar to BY REFERENCE, except that the address of a copy of the data item is sent. This has the effect that any changes made to the parameter in the called program are not seen by the caller. Starting with Version 2.0, literals passed as parameters are automatically passed BY CONTENT.

   The BY VALUE method causes the contents of the data item to be passed to the receiving program (the actual data, not its address). This may *not* be specified if the receiving program is a COBOL program (results are undefined in this case). This method is typically used to pass numeric data to C subprograms. For optimal portability we recommend that parameters being passed by this method be level 01 or 77 items described as SIGNED-INT, UNSIGNED-INT, SIGNED-SHORT, UNSIGNED-SHORT, SIGNED-LONG, UNSIGNED-LONG, or POINTER.

   When the name of an open COBOL file is given as a parameter, the operating system's file handle is passed. One possible reason for doing this is to call an operating system function that allows the file to retrieve some information that is not available through COBOL. Several special rules apply to this usage. See rules 30 through 35 below.

7. The NULL and OMITTED options are synonymous. They allow you to skip a parameter in a USING phrase. The corresponding parameter in the called program must not be referenced. If the called program is a C program, then a NULL value is passed to the corresponding parameter.

8. Index names referred to in the Linkage Section of the called program do not correspond to any index names in the calling program. Index data items are always stored in the local memory of their programs.

9. If the called program does not have the INITIAL attribute, then it is in its initial state the first time it is called, and the first time it is called after a CANCEL verb has referred to it. On all other calls, the program is in the same state as when it last exited.

10. If the called program has the INITIAL attribute, it is in its initial state every time it is called.

11. Files contained in the called program are not open whenever the called program is in its initial state. Otherwise, the status and positioning of the contained files is the same as when the program last exited.

12. If the ON EXCEPTION phrase is present and the called program cannot be initiated, *statement-1* executes. If the called program cannot be initiated when there is no EXCEPTION phrase, a runtime error occurs and the program halts. A program cannot be initiated if *program-name* cannot be resolved using the rules described in *ACUCOBOL-GT User's Guide* section 2.9, "Calling Subprograms."

13. If the NOT ON EXCEPTION phrase is present and the called program is successfully initiated, *statement-2* executes when the called program returns.

14. The ON EXCEPTION and NOT ON EXCEPTION phrases execute in the original (parent) thread. The parent thread suspends long enough to determine whether or not the CALL will succeed. This allows it to determine whether to execute the EXCEPTION or NOT EXCEPTION case.

15. A called program's file state and data items are distinct for each thread that exists at the time of the call (including the thread created by the CALL THREAD). Thus, if *thread-1* calls *program-a*, and *thread-2* calls *program-a*, there will be two copies of the data from *program-a* in memory, one set for each thread. Threads created in the called program, or any of its subprograms, share the called program's data and file state.

16. The CALL PROGRAM verb is similar to the CHAIN verb. It causes the current run unit to terminate and initiates a new run unit. However, USING parameters are passed to data items specified in Linkage, just as they are for a standard CALL verb.

Some (incorrect) usages of the CALL statement USING parameters from the Linkage section of the called program can result in so-called "intermediate" runtime errors that call installed error procedures. There currently are two such related errors: "Use of a LINKAGE data item not passed by the caller," and "Passed USING item smaller than corresponding LINKAGE item." A passed USING item that is larger than the corresponding Linkage item does not generate an error.

Another runtime error can occur when the CALL statement attempts USING parameters of invalid structure or missing parameters. That kind of error is announced as "Invalid or missing parameter," and it is also an "intermediate" type of runtime error that calls installed error procedures.

See Book 4 *Appendices*, Appendix I "Library Routines," for detailed discussion of the runtime error and exit procedures.

17. When a CALL PROGRAM verb succeeds, all program switches are set to their "off" state. You can specify switches to be set "on" in the CALL PROGRAM statement. You accomplish this by specifying the switch names immediately after the program name in the statement. Each switch name must be alphabetic and must be preceded by a slash (/). For example, to turn on switches "A" and "B" you could use this statement:

    ```
    CALL PROGRAM "MYPROG/A/B" USING ...
    ```

18. If the CALL PROGRAM verb cannot find the called program, and no EXCEPTION phrase has been specified, control passes to the next executable statement.

19. The CALL PROGRAM verb accepts the presence of routines whose name begins with a "#" sign. When one of these routines is specified, the CALL PROGRAM statement is ignored. In ICOBOL, routines with these names are special-purpose system routines.

20. If the RUN option is used, then *program-name* refers to the main program of another run unit. This run unit is initiated and continues until it executes a STOP RUN. At that point, control is returned to the next executable sentence in the calling program. Parameters are passed to the called run unit in the same manner as specified for the CHAIN statement.

The called run unit is logically distinct from the calling run unit. It may call programs that are active in the calling unit without generating an error. External areas are associated with a logical run unit, so that a new run unit does not have access to external areas created by the calling run unit. Take care that file records locked by the calling run unit do not interfere with the called run unit, because there will be no opportunity for the caller to release its records while the called unit is active.

Type-ahead is retained both when the new run unit is called and upon return from it.

The precise amount of memory required for the runtime varies by platform and is affected by configuration variables such as **V_BUFFERS** and by any products or C routines you have linked in. When the CALL RUN executes, the memory associated with the new run unit is allocated, and the old run unit (with its attendant memory) remains active. Memory allocated by the runtime for program code is freed automatically at the STOP RUN in the called program. Memory used for open windows and V_BUFFERS is not freed until the runtime exits. You should explicitly free any memory that you allocated with the **M$ALLOC (Dynamic Memory Routine)** otherwise this memory is not freed until the runtime exits.

21. A special register named RETURN-CODE is automatically created by the compiler and is shared by all programs of a run unit. This special register is defined as:

```
77  RETURN-CODE  SIGNED-LONG, EXTERNAL.
```

If you call a C program via the “direct” interface, the return value of the C function is placed into this register. If you call the SYSTEM library routine, the status of the call is placed into this register. The verbs EXIT, STOP, and GOBACK can also place a value into the RETURN-CODE register.

The compiler also creates an unsigned version of the return code called RETURN-UNSIGNED. It has the following implied definition:

```
77  RETURN-UNSIGNED
    REDEFINES RETURN-CODE
    UNSIGNED-LONG, EXTERNAL.
```

22. If the RETURNING phrase is used, then the "return value" of the called program is moved to *return-val*. This is accomplished by the following rule:

    a. If *return-val* is a signed data item, then the value of RETURN-CODE is moved to *return-val* according to the rules of the MOVE statement.

    b. If *return-val* is unsigned, then RETURN-UNSIGNED is moved instead.

23. You should avoid using RETURN-CODE or RETURN-UNSIGNED for *return-val*. This is pointless because, after assigning a value to *return-val*, the CALL statement restores the previous value of RETURN-CODE.

24. On Windows systems, if you CALL a DLL, the runtime assumes that the DLL returns a "long" (at least 32 bits of data). This ensures that all of the data returned from the DLL is captured. However, in situations where the DLL returns less than 32 bits of data (such as a DLL that returns a "short"), some of the data might be random. To get the right size for the return value of DLLs, use the RETURNING phrase and set the receiving variable to the same size as the DLL's return value. The easiest way to do this is to declare the receiving variable to be the same type as the DLL's return type. For example, if the DLL "MYDLL" returns an "int," you could do the following:

```
77  RETURN-VAL       SIGNED-INT.
CALL "MYDLL" RETURNING RETURN-VAL.
```

25. After *return-val* is set, RETURN-CODE is set to the value it had immediately prior to the CALL statement. Thus the called program does not affect the value of RETURN-CODE in the calling program.

26. If the CALL statement fails with an exception, then *return-val* is not updated.

27. You may pass floating-point data to subroutines normally with the CALL verb. Note that you may not pass a floating-point item BY VALUE. This restriction exists for portability reasons (some machines pass floating-point using a convention different from that used for integer items). You should pass floating-point items BY REFERENCE. This will pass a pointer to the item, which the receiving routine can then retrieve by "de-referencing" the pointer.

28. A program may directly or indirectly call itself. A CALL statement that calls the active program (itself) is a *recursive call*. For more information, see the **RECURSION** configuration variable in Appendix H, Book 4, *Appendices*. For information on sharing data in recursively called programs (such as in the HP COBOL environment), see the **RECURSION_DATA_GLOBAL** configuration variable. See also **Section 2.10.1** in Book 1, *ACUCOBOL-GT User's Guide*.

29. Errors that occur as a result of the CALL statement USING parameters from the Linkage section of the called program belong to an "intermediate" type of runtime errors that call installed error procedures. There are two such errors: "Use of a LINKAGE data item not passed by the caller," and "Passed USING item smaller than corresponding LINKAGE item."

Passing file handles

1. For compatibility with HP COBOL, a file handle is automatically passed BY VALUE unless it is immediately preceded by a BY REFERENCE or BY CONTENT specification. File handles passed to COBOL subroutines must be preceded with BY REFERENCE or BY CONTENT because COBOL routines cannot take BY VALUE parameters.

2. If the called subroutine is a COBOL routine, the handle passed is PIC S9(4) COMP-5. You can override this with the compiler option "--fileIdSize=#" where "#" is either "2", "4", or "8" to specify the number of bytes you want in the passed integer.

3. If the called subroutine is not COBOL, the handle is passed as a signed native integer using the host's default integer size.

4. The file handle passed is the host file system's identifying value for the open file. For all current implementations, this is the value returned by the C "open" function. If the host file system does not have this information available, then "-1" is used instead. This can happen if the host system is not a file (e.g. Acu4GL for Oracle) or the host system does not provide a way of obtaining the handle (e.g. C-ISAM interface). Files served by AcuServer® also use "-1" because there is no useful way to use a remote process' open file handle.

5. For Vision files in the multi-file format (Version 5 and 4), the handle used is the handle of the first data segment (this is the same file used when opening the file).

6. It is best to avoid performing actual I/O on the file using this file handle because the COBOL file system will be unaware of any state changes to the file and may perform incorrectly. It is possible to corrupt data this way.

CANCEL Statement

The CANCEL verb places a program into its initial state.

General Format

Format 1

```
CANCEL {program-name} ...
```

Format 2

```
CANCEL ALL
```

Format 3

```
CANCEL SORT
```

Syntax Rule

Program-name is a nonnumeric literal or an alphanumeric data item. It may not be an ALL literal.

General Rules

1. After a CANCEL verb executes, the affected programs are placed into their initial states. This closes any open files contained in the canceled program and ensures that any VALUE clauses are in effect when those programs are called again. By default, memory used by the programs is released. If the mechanism for *logical cancels* is enabled, the programs are cached in memory. For information about the effects and use of logical cancels, see section 6.3, “Memory Management,” in Book 1.

2. In Format 1, each *program-name* refers to the CALL name of a program to cancel. For more information, see the entry in this section for the “CALL Statement,” and Book 1, *ACUCOBOL-GT User’s Guide*, section 2.9, “Calling Subprograms.”

3. In Format 2, every program that has been called but is not active is canceled. A program is active if it has been called (or is the initial program of the run) and has not yet exited.

Note: The CANCEL_ALL_DLLS configuration variable can be used to exclude DLLs and shared object libraries from the results of the CANCEL ALL statement. See the listing for **CANCEL_ALL_DLLS** in Appendix H for details.

---

4. A CANCEL statement is ignored if it refers to an active program. A CANCEL statement may also refer to a program that has never been active. Such a reference has no effect.

5. A CANCEL statement has no effect on a program that has a RESIDENT phrase in its PROGRAM-ID paragraph.

6. When you cancel a program that exists in more than one thread, it is canceled only in the current thread. CANCEL ALL cancels only programs in the current thread. If a program is active in any thread that it shares data with, the CANCEL will have no effect.

7. In Format 3, any active sort is terminated. Only one sort may be active at a time; using Format 3 guarantees that no sort is active, and thus prepares you to initiate a new sort safely.

CHAIN Statement

The CHAIN statement provides a method for starting another run unit.

General Format

```
CHAIN program-name [ USING {parameter} ... ]

   [ ON {EXCEPTION} statement ]
        {OVERFLOW }

   [ END-CHAIN ]
```

Syntax Rules

1. *Program-name* is a nonnumeric literal or an alphanumeric data item.

2. *Parameter* is any non-level 88 data item or a nonnumeric literal. No more than 50 *parameters* may be specified.

3. *Statement* is an imperative statement.

4. If a CHAIN statement appears in a consecutive sequence of imperative statements within a sentence, it must be the last statement in that sequence, unless the EXCEPTION phrase is specified.

General Rules

1. The CHAIN statement causes the current run unit to terminate and initiates a new run unit. Executing a CHAIN statement has the following effects:

   a. The current run unit is halted as if a STOP RUN statement were executed.

   b. The run unit specified by *program-name* is initiated. *Program-name* is resolved into an executable program name using the same rules specified for the CALL statement.

2. If the EXCEPTION phrase is used, then *statement* executes if the CHAIN statement is unable to load *program-name*. This is usually caused by *program-name* being either absent or not readable by the runtime system. Certain types of errors cannot be caught by the ON EXCEPTION phrase because of the nature of the CHAIN statement. If the EXCEPTION phrase is not present when an error occurs, the runtime system prints a message and halts.

3. If the USING phrase is specified, each *parameter* is transferred to the new run unit. Each *parameter* is mapped to the corresponding CHAINING argument in the new run unit. See **section 6.5, "Procedure Division Format,"** for a description of the CHAINING phrase.

CLOSE Statement

The CLOSE statement is used to terminate the processing of a file or files, to change the current video-terminal window, or to close a floating window.

General Format

Format 1

```
CLOSE { file-name [REEL] [ WITH {NO REWIND} ] } ...
                  [UNIT]        {LOCK     }
```

Format 2

```
CLOSE WINDOW window-handle [ WITH NO DISPLAY ]
```

Syntax Rules

1. *File-name* must be the name of a file described in the Data Division. It may not be a sort file.
2. If the REEL or UNIT option is used, then *file-name* must refer to a sequential file.
3. *Window-handle* can be either a PIC X(10) data item or a HANDLE data item. If used with a SUBWINDOW, *window-handle* must have been the object of a POP-UP AREA phrase in a DISPLAY statement.

General Rules

Format 1 (CLOSE File)

1. A file referred to by a CLOSE statement must be in the open state. After the execution of a successful CLOSE statement, each *file-name* is in the closed state. All record and file locks held by those files are released.
2. Any FILE STATUS variables associated with the *file-names* are updated by the CLOSE verb.
3. The NO REWIND option lets you hold a pipe open when closing its corresponding file. This allows you to gather multiple reports into a single job for the print spooler. To use NO REWIND in this way, create a configuration file entry as described in Appendix H, Book 4, *Appendices* under **SPOOL_FILE**.
4. The LOCK option prevents the affected files from being opened again by this run unit in its current execution.

5. If the UNIT or REEL option is specified, then the file remains open and the next reel of the multi-reel file is mounted. Since ACUCOBOL-GT does not directly support multi-reel files, specifying this option causes the CLOSE statement to have no effect other than to update the FILE STATUS variable.

Format 2 (CLOSE WINDOW)

1. The CLOSE WINDOW verb is used to remove floating windows and subwindows.

2. *Window-handle* must be a handle returned by a DISPLAY FLOATING WINDOW statement, or the object of a POP-UP phrase of a DISPLAY WINDOW statement that executed in the current run unit. Furthermore, it cannot have been the object of a CLOSE WINDOW verb nor may it have been otherwise modified.

3. The CLOSE WINDOW verb restores the contents of the screen covered by the window that is being destroyed. In other words, the window that was created by that DISPLAY FLOATING WINDOW or DISPLAY WINDOW statement is removed from the screen and replaced by the contents of the screen that was "under" that window. Any memory associated with the closed window is then freed.

4. The window that was current when the terminating window was created becomes the active window. The cursor is positioned to the location it occupied when the window was created.

5. The WITH NO DISPLAY option causes the closed window to remain on the screen. The effect is the same as a normal CLOSE WINDOW verb, except that no updating of the console takes place. This can be used to free memory used by a window when you do not want to physically remove it from the screen. For example, suppose you want to remove a series of nested windows and want to reduce the amount of screen activity that occurs. In this case, you can close all of the nested windows with the NO DISPLAY option and then close the outermost window in the normal manner.

   WITH NO DISPLAY is ignored when a floating window is closed. It is effective only when pop-up windows are closed.

6. CLOSE WINDOW *window-handle* is synonymous with DESTROY *window-handle*.

COMMIT Statement

The COMMIT statement unlocks locked records and (optionally) flushes buffers to disk. COMMIT may also indicate the end of a transaction, and cause the changes to be written to the transaction log file.

General Format

```
COMMIT TRANSACTION
```

General Rules

1. When this statement is executed, all locked records owned by the current run-unit are unlocked.

2. COMMIT also causes a request to be made to the host operating system to flush all buffers to disk. The exact effect of flushing disk buffers depends on the host operating system. Some systems do not ensure that buffers are fully flushed when the COMMIT verb finishes. For example, UNIX only schedules a flush, which occurs over the next few seconds. VAX/VMS already flushes after each write (except for locked files).

3. You can prevent the buffers from being flushed to disk by using the FLUSH-ON-COMMIT line in your runtime configuration file.

4. The function of COMMIT depends on whether or not it ends a transaction. The following rules describe how transaction management operates with Vision and relative files. For other file systems linked with the runtime, each system's native mechanism for transaction management is invoked. See the interface document for the specific file system for more details.

5. When ROLLBACK is enabled in the FILE-CONTROL entry for a file, the record and file locking rules are extended for that file. Every record updated as part of a transaction is locked until that transaction is committed or rolled back. The COMMIT verb removes these locks. Record locks applied when the file is read are kept until the end of the transaction, if ROLLBACK is enabled for the file.

   Record locks are held during a transaction in order to prevent another process from updating the records in a way which might make rollback impossible. Note, however, that a record may be deleted during a

transaction, and another process is allowed to write a record with the same record key to the file. If this happens, and duplicates are disallowed on that record key, then the ROLLBACK will fail with a duplicate key error.

6. During a transaction involving Vision or relative files, a CLOSE of a file that is locked, or that has locked records, is postponed until the transaction is committed or rolled back.

   If the same physical file is opened again *within the transaction*, even if a different logical file (different SELECT) is used, the postponed CLOSE is canceled. Note that the mode of the original OPEN is retained. (For example, if the file were originally OPEN I-O, and if the CLOSE were canceled, then an OPEN OUTPUT on the same file within the same transaction would *not* empty the file.) When the second OPEN is encountered, the file position is reset to the beginning so that a READ NEXT would read the first file in the record. CLOSE is handled in this special way so that record locks are held--these locks are necessary for rollback.

7. COMMIT locks the log file, checks its integrity, then writes the changes and unlocks the log file. The log file is specified with the **filename_LOG** or **LOG_FILE** configuration variable. See *ACUCOBOL-GT User's Guide*, Chapter 5, **Section 5.1.6**.

8. There is an implicit COMMIT before a STOP RUN or before the end of the program, unless the **STOP_RUN_ROLLBACK** configuration variable is set to 1. Then, an implicit ROLLBACK occurs.

COMPUTE Statement

The COMPUTE statement provides the ability to perform general arithmetic computations.

General Format

```
COMPUTE { result [ROUNDED] } ... { =     } arithmetic-expr
                                 { EQUAL }

   [ ON SIZE ERROR statement ]

   [ NOT ON SIZE ERROR statement ]
```

```
[ END-COMPUTE ]
```

Syntax Rules

1. *Result* is an elementary numeric or numeric-edited data item.
2. *Arithmetic-expr* is any arithmetic expression.
3. *Statement* is an imperative statement.

General Rules

1. The *arithmetic-expr* is evaluated and assigned to each *result* variable.
2. Additional information can be found in the sections covering Arithmetic Operations (6.4.1), Multiple Receiving Fields (6.4.2), the ROUNDED option (6.4.3), and the SIZE ERROR option (6.4.4).

CONTINUE Statement

The CONTINUE statement is a “no action” statement.

General Format

```
CONTINUE
```

General Rule

The CONTINUE statement performs no actions. It can be used in any case where a statement is required but no action is wanted.

CREATE Statement

The CREATE statement creates a new instance of a non-graphical object such as a COM object or .NET assembly. Use it specifically for objects or assemblies that are not visual in nature. Use the Screen Section or DISPLAY statement to create an instance of a graphical .NET assembly or an ActiveX control.

Note: CREATE can be used with thin client applications to create instances of an object on the client or on remote Windows servers. CREATE cannot be used to create an object on a non-Windows server such as UNIX or VMS; however, non-Windows servers running AcuConnect can provide connectivity to Windows servers in a multitiered configuration.

The formats of the CREATE statement include:

- Format 1: CREATE object-name
- Format 2: CREATE assembly-name

General Format

Format 1

CREATE object-name creates a COM object.

```
CREATE object-name

HANDLE { IN } object-handle
       { IS }
```

Remaining phrases are optional.

```
SERVER-NAME { IS } server-name
            { =  }
LICENSE-KEY { IS } license-key
            { =  }
EVENT PROCEDURE IS { proc-1 [ {THROUGH} proc-2 ] }
                             {THRU }
                   { NULL }
```

Format 2

CREATE assembly-name creates a non-graphical .NET object.

```
CREATE "assembly-name"

NAMESPACE { IS } "namespace"
```

```
CLASS-NAME { IS } "class-name"

HANDLE { IN } object-handle
       { IS }
```

Remaining phrases are optional.

```
VERSION { IS } "version"

CULTURE { IS } "culture"

STRONG-NAME { IS } "strong-name"

CONSTRUCTOR { IS } CONSTRUCTOR[n] parameters...

MODULE { IS } "module"

FILE-PATH { IS } "file-path"
```

Syntax Rules

1. *Object-name* is an optionally qualified name of a COM object defined in the special-names paragraph.
2. *Server-name* is an alphanumeric literal or data item.
3. *License-key* is an alphanumeric literal or data item.
4. *Object-handle* is a USAGE HANDLE data item.
5. *Proc-1* and *proc-2* are procedure names relating to COM events. (For more information on COM events and event procedures, see section 4.4 in *A Guide to Interoperating with ACUCOBOL-GT.*)
6. Any value surrounded by quotation marks is an alphanumeric literal and is case-sensitive.

General Rules

Format 1 (CREATE object-name)

1. In a non-Thin Client environment the COM object is registered and instantiated on the same machine the runtime is executing on. In a Thin Client scenario the COM object is registered and instantiated on the client machine rather than the server where the runtime is executing.

2. Server-name identifies a remote machine on which to create and execute the COM object. It can be specified as a UNC, DNS name, or IP address. Server-name must be the name of a Windows machine. CREATE cannot create objects on non-Windows servers. When a server-name is provided - the COM object's interface is instantiated on the machine where the application is executing and the back-end is instantiated on the specified server where the work is done and resources can be accessed. In this case the COM object must be registered on both machines.

3. In a Thin Client environment the customer may also specify the Server-name with the prefix, Local: e.g. CREATE COM_object SERVER-NAME "Local:the_server_name". In this case the COM object is wholly instantiated on the specified server where the COM object is registered, all the work is done and all resources must reside.

4. Some COM objects are licensed for run time using a license key that is provided to you by the object vendor. This license key is a text string usually located in a ".lic" file. By setting the value of the LICENSE-KEY property to this license key, you embed this license key in your COBOL program. Then when you run your COBOL program, the license key is passed to the COM object for verification. You do not send the ".lic" file or license key to the end-user. Set LICENSE-KEY when you create the object (i.e., in the CREATE statement). The default value is " ". When LICENSE-KEY is " " (i.e., the default) and the COM object supports the licensing mechanism, the control performs its own license verification. Some objects require a ".lic" file to do this. Others may check the system registry or hard disk for other properly installed and licensed software.

Format 2 (CREATE assembly-name)

1. Literal values for assembly parameters are located in the COPY file generated by the NETDEFGEN utility. The same COPY file must be included in the SPECIAL-NAMES paragraph of your program.

2. *Assembly-name* is the name of a .NET assembly defined in the NETDEFGEN COPY file. This must be the DLL name of a non-graphical control, not an executable file. Non-graphical controls are generated by Visual Studio when a developer selects a "Class Library" project type.

3. *Namespace* is a NameSpace in the assembly.

4. *Class-name* is a class in the NameSpace.

5. *Version* is the version number of the assembly.

6. *Culture* is cultural information related to the assembly.

7. *Strong-name* is the cryptographic key required to access the assembly, if any. If the assembly requires such a key, as all assemblies in the Global Assembly Cache (GAC) do, it is shown in the COPY file under the keyword STRONG.

8. All classes that result in an object have a CONSTRUCTOR, which is a sort of method. If you see a CONSTRUCTOR identifier in the COPY file without a parameter list, the field may be omitted from your COBOL program. If all listed CONSTRUCTORs have parameters, then you must choose which CONSTRUCTOR and parameters to use. *Constructor*(n) is the constructor that you want to use followed by its parameter data.

9. *Module* identifies a file where a combination of NameSpaces and Classes resides. It is used when the assembly is constructed of other assemblies.

10. *File-path* is the path of an XML file, and that XML file contains the path where the .NET assembly is located. Use FILE-PATH when the assembly that you want to access does not reside in the GAC or in the same directory as "wrun32.exe". Assemblies that reside in the GAC will have the STRONG keyword in the NETDEFGEN COPY file.

DELETE Statement

The DELETE statement logically removes a record from a file or a file from the host system.

General Format

Format 1

```
DELETE file-name RECORD

   [ INVALID KEY statement-1 ]

   [ NOT INVALID KEY statement-2 ]

   [ END-DELETE ]
```

Format 2

```
DELETE FILE file-name
```

Syntax Rules

1. *File-name* must refer to a file described in the Data Division. It may not be a sort file. In Format 1, it cannot be the name of a sequential file.
2. *Statement-1* and *statement-2* are imperative statements.
3. The INVALID KEY and NOT INVALID KEY clauses must not be specified for a DELETE statement that references a file that is in the sequential access mode.

General Rules

Format 1 - DELETE RECORD

1. The file must reside on a mass-storage device and be open in the I/O mode when the DELETE statement executes.
2. For files in the sequential access mode, the last I/O statement executed for the file prior to the execution of the DELETE statement must have been a successful READ statement. The DELETE statement deletes that record from the file.

3. For a relative file in the random or dynamic access mode, the DELETE statement removes the record identified by the file's RELATIVE KEY data item. If that record does not exist, an invalid key condition exists.

4. For an indexed file in the random or dynamic access mode, the DELETE statement removes the record identified by that file's RECORD KEY data item. If that record does not exist, an invalid key condition exists.

5. After a successful DELETE, the deleted record has been logically removed from the file and may no longer be accessed.

6. The execution of the DELETE statement does not affect the file's record area.

7. The current file position is not changed by the DELETE statement.

8. The FILE STATUS variable of the file is updated.

9. If there is an applicable USE AFTER EXCEPTION procedure, it executes whenever a non-zero I/O status applies to the DELETE. However, it does not execute if an invalid key condition exists and the INVALID KEY phrase is used. If a USE AFTER EXCEPTION procedure would normally execute, but none has been specified, then the program prints an error message and halts.

10. *Statement-1* is executed if it is specified and the invalid key condition exists.

11. *Statement-2* is executed if it is specified and the DELETE statement is successful.

Format 2 - DELETE FILE

1. The DELETE FILE statement removes a file from the host computer system. The file named by *file-name* must be closed and reside on a mass-storage device.

2. With one exception, the organization and record layout of *file-name* need not match the file being deleted, although it is highly recommended that they do. For Vision Version 5 and 4 files, the organization and record layout of *file-name* must match the file being deleted.

3. If you match the type of the file referenced (sequential, relative, or indexed) with the type of the file being removed, your programs will consistently work with versions of ACUCOBOL-GT based on different file systems. For example, C-ISAM and Vision Versions 5 and 4 implement indexed files using two physical disk files for each COBOL indexed file. An ACUCOBOL-GT runtime using Vision Version 5, 4, or C-ISAM needs to know if an index file is being removed so that it will look for both physical files when an indexed file is removed.

   When the DELETE statement executes, *file-name* is translated as for the OPEN statement, and the corresponding file is removed from the system.

4. The FILE STATUS data item is updated by the DELETE FILE statement.

DESTROY Statement

The DESTROY statement eliminates windows, controls, threads, fonts, layout managers, and menus.

Note: The DESTROY verb cannot destroy the main application window. If you attempt to use the DESTROY verb to destroy the main application window, the statement is ignored. The main application window is destroyed automatically when the program terminates.

General Format

Format 1

```
DESTROY  { screen-name-1 } ...
         { handle-1      }
```

Format 2

```
DESTROY ALL CONTROLS
```

Format 3

```
DESTROY { CONTROL
```

Remaining phrases are optional and can appear in any order.

```
AT screen-loc  [CELL  ]
               [CELLS ]
               [PIXEL ]
               [PIXELS]

AT LINE NUMBER line-num  [CELL  ]
                         [CELLS ]
                         [PIXEL ]
                         [PIXELS]

AT {COLUMN  } NUMBER col-num  [CELL  ]
   {COL     }                 [CELLS ]
   {POSITION}                 [PIXEL ]
   {POS     }                 [PIXELS]

AT CLINE NUMBER cline-num   [CELL ]
                            [CELLS]

AT CCOL NUMBER ccol-num   [CELL ]
                          [CELLS]
  } . . .
```

Syntax Rules

1. *Screen-name-1* is the name of a screen description entry found in the Screen Section.
2. *Handle-1* is a USAGE HANDLE or PIC X(10) data item.
3. *Screen-loc* is an integer data item or literal that contains exactly 4 or 6 digits.
4. *Line-num*, *col-num*, *cline-num* and *ccol-num* are numeric data items or literals. These may contain non-integer values.
5. Format 1 and Format 3 statements can be mixed into a single DESTROY statement.

General Rules

1. The DESTROY verb removes from memory the items it acts on. It also removes those items from the screen if that is appropriate. The exact meaning of the DESTROY verb depends on the type of item it is acting upon, as described below.

2. If the statement is a Format 1 DESTROY *handle-1* statement, the handle is valid, and the DESTROY statement succeeds, *handle-1* is set to the value NULL (binary zeros).

3. If *handle-1* contains a valid handle to a control, then the control is removed from the screen (if it is visible) and any memory associated with the control is released.

4. If *handle-1* contains the valid handle of a floating window or subwindow, the DESTROY statement acts the same as a CLOSE WINDOW statement acting on *handle-1* (with no additional options), except that DESTROY sets the value of *handle-1* to NULL. When you destroy (close) a floating window, every control contained in that window is also destroyed. Also, any other floating windows that are children of that window are destroyed.

   Note: Attempting to DESTROY the main application window has no effect. However, this behavior is subject to change in a future release and should not be relied on.

   ---

5. If *handle-1* is a handle of a font, that font is removed from memory. If *handle-1* is a standard font (that is, one retrieved by ACCEPT from STANDARD OBJECT), destroying it has no effect. Destroying a font that is actively in use by a control or window is an error with unpredictable consequences. If you are destroying a set of controls and their associated fonts, you should destroy the controls first, and then destroy their fonts.

6. If *handle-1* is a handle to a thread, "DESTROY *handle-1*" has the same affect as a "STOP THREAD *handle-1*" statement, except that the DESTROY statement also sets the value of *handle-1* to NULL.

7. If *handle-1* is the handle of a menu, then that menu is destroyed, and any memory associated with it is freed. It is an error (not caught) to destroy a menu that is currently associated with a window or a control. A menu that is displayed as the menu bar of a window is automatically destroyed when the corresponding window is destroyed. A menu displayed as the menu bar of a window should not be explicitly destroyed by your program (unless you remove it from the window first). Menus associated with windows or controls as pop-up menus

are not automatically destroyed when the corresponding window or control is destroyed. This makes it easier to use the same pop-up menu in more than one control or window.

8. If *handle-1* is a handle to a layout manager, the layout manager is removed from memory. A layout manager that is actively being used by a window should not be destroyed. You should first destroy the window or remove the layout manager from the window.

9. If *handle-1* contains a value other than a valid handle to a window, control, thread, font, menu, or layout manager, the DESTROY verb has no effect.

10. If *screen-name-1* is a Format 2 screen description entry (i.e., it describes a control), that control is removed from the screen, and any memory associated with it is released. *Screen-name-1* no longer refers to an existing control.

11. If *screen-name-1* is an elementary Format 1 screen description entry (i.e., it describes a textual field), the DESTROY verb has no effect.

12. If *screen-name-1* is a group item, each subsidiary elementary item is destroyed (i.e., the controls are destroyed and the textual fields are left alone).

13. A Format 2 DESTROY statement (DESTROY ALL CONTROLS) destroys every control contained in the current window. This includes controls created by Screen Section entries as well as controls created directly with a DISPLAY Control verb.

14. A Format 3 DESTROY statement destroys the control located at the screen position specified by the AT, LINE, and COLUMN phrases in the current window (on non-graphical systems, the CLINE and CCOL phrases also apply). The runtime system maintains a list of controls in each window. When attempting to destroy a control at a specific location, the runtime searches this list, using the first control it finds that exactly matches the location. The list is maintained in the order in which the controls are created. If the runtime does not find a control at the specified location, then nothing happens.

15. Use Format 1 “DESTROY *handle*” to destroy non-graphical .NET controls or assemblies. For graphical .NET controls, you can use either Format 1 “DESTROY *handle*” or Format 2 “DESTROY ALL”.

DISPLAY Statement

The DISPLAY statement provides for low-volume output from the program. The formats of the DISPLAY statement include:

- Format 1: **DISPLAY src-item**
- Format 2: **DISPLAY screen-name**
- Format 3: **DISPLAY WINDOW**
- Format 4: **DISPLAY SCREEN SIZE**
- Format 5: **DISPLAY LINE**
- Format 6: **DISPLAY BOX**
- Format 7: **DISPLAY UPON WINDOW TITLE**
- Format 8: **DISPLAY UPON COMMAND LINE**
- Format 9: **DISPLAY src-item (ANSI format)**
- Format 10: **DISPLAY UPON GLOBAL TITLE**
- Format 11: **DISPLAY FLOATING WINDOW**
- Format 12: **DISPLAY INITIAL WINDOW**
- Format 13: **DISPLAY TOOL-BAR**
- Format 14: **DISPLAY control-type-name**
- Format 15: **DISPLAY MESSAGE BOX**
- Format 16: **DISPLAY external-form-item**
- Format 17: **DISPLAY UPON ENVIRONMENT-NAME**
- Format 18: **DISPLAY assembly-name**

Note: Different formats of the DISPLAY statement may be mixed together in one DISPLAY statement, as long as no ambiguity results. The effect is the same as specifying each DISPLAY statement separately.

DISPLAY src-item

Format 1

DISPLAY src-item displays an individual field to the screen.

```
DISPLAY { {src-item}
          {OMITTED }

 [ UPON new-window ]
```

Remaining phrases are optional, can appear in any order.

```
AT screen-loc

AT LINE NUMBER line-num

AT {COLUMN  } NUMBER col-num
   {COL     }
   {POSITION}
   {POS     }

WITH SIZE length

WITH NO ADVANCING

{ERASE} [TO END OF] {LINE  }     (VAX, ICOBOL)
{BLANK}             {SCREEN}

{ERASE} [EOS]                    (RM)
{BLANK} [EOL]

WITH  {BELL}
      {BEEP}

{UNDERLINED}
{HIGHLIGHT }
```

```
{HIGH      }
{BOLD      }
{LOWLIGHT  }
{LOW       }
{STANDARD  }

WITH  {BLINKING}
      {BLINK   }

{REVERSE-VIDEO}
{REVERSE      }
{REVERSED     }

SAME

WITH {COLOR } color-val
     {COLOUR}

{FOREGROUND-COLOR } IS fg-color
{FOREGROUND-COLOUR}

{BACKGROUND-COLOR } IS bg-color
{BACKGROUND-COLOUR}

SCROLL [UP  ] [ BY scrl-num {LINE } ]
       [DOWN]               {LINES}

OUTPUT {JUSTIFIED}  {LEFT    }
       {JUST     }  {RIGHT   }
                    {CENTERED}

WITH {CONVERSION}
     {CONVERT   }

CONTROL cntrl-string

UPON CRT

} ...
```

Syntax Rules

1. Different formats of the DISPLAY statement may be mixed together in one DISPLAY statement, as long as no ambiguity results. The effect is the same as specifying each DISPLAY statement separately.
2. *Src-item* is a literal or data item. It must be USAGE DISPLAY unless the CONVERSION phrase is also specified. *Src-item* specifies the data to be displayed.
3. *New-window* is a USAGE HANDLE or PIC X(10) data item.
4. *Screen-loc* is an integer data item or literal containing exactly 4 or 6 digits. It may also be a group item of 4 or 6 characters. If a numeric item is used, it must be a non-negative integer.
5. *Line-num*, *col-num*, and *length* are numeric data items or literals. They may be non-integer values. You can also specify the value with an arithmetic expression.
6. *Color-val* and *scrl-num* are numeric data items or literals. *Color-val* can also be an arithmetic expression, except when used in the Screen Section.
7. *Fg-color* and *bg-color* are integer literals or numeric data items. They may be arithmetic expressions. See **section 6.4.9**, FOREGROUND-COLOR and BACKGROUND-COLOR phrases, for a more detailed discussion of color settings and values.
8. *Cntrl-string* is a nonnumeric literal or data item.
9. If the UPON phrase is used it must be the first optional phrase specified.
10. If the AT phrase is specified, neither the LINE nor the COLUMN phrase may be specified.
11. If the COLOR phrase is specified, neither the FOREGROUND-COLOR nor the BACKGROUND-COLOR phrase may be specified.
12. IS and “=” are synonymous.
13. COLUMN, COL, POSITION, and POS are equivalent.

14. BELL and BEEP are equivalent.
15. BLANK and ERASE are equivalent.
16. HIGHLIGHT, HIGH and BOLD are synonymous.
17. LOWLIGHT and LOW are equivalent.
18. UNDERLINE and UNDERLINED are equivalent.
19. BLINK and BLINKING are equivalent.
20. REVERSE-VIDEO, REVERSE, and REVERSED are equivalent.
21. CONVERT and CONVERSION are equivalent.
22. COLOR and COLOUR are synonymous.
23. FOREGROUND-COLOR and FOREGROUND-COLOUR are synonymous.
24. BACKGROUND-COLOR and BACKGROUND-COLOUR are synonymous.
25. The ERASE phrase has two forms. In VAX COBOL and ICOBOL compatibility modes, the ERASE SCREEN/LINE form must be used. In RM/COBOL mode, the ERASE EOS/EOL form must be used. This is indicated in the General Format by the symbols "*(VAX, ICOBOL)*" and "*(RM)*"
26. You may add "UPON CRT" to a Format 1 DISPLAY statement to distinguish it from a Format 9 DISPLAY statement. You need to do this only if you use the "-Ca" compiler option.

General Rules

1. The DISPLAY statement sends each of its *src-items* to the video terminal attached to the executing program. If more than one *src-item* is specified, each is treated as if it were in a separate DISPLAY statement in the order listed. Note, however, in VAX COBOL and ICOBOL compatibility mode, NO ADVANCING is automatically implied for each *src-item* except for the last one.

2. The precise action of the DISPLAY statement depends both on the various clauses specified and the *compatibility mode* that the compiler is run in. These actions are detailed in the following rules.

3. The effects of the various optional phrases are described in **section 6.4.9, "Common Screen Options."**

4. If the OMITTED option is used, then no *src-item* is sent to the screen. This can be used to cause the action of various optional phrases without sending any actual data. For example, "DISPLAY OMITTED, BELL" will cause the terminal's bell to ring. If a SIZE phrase is specified, then the OMITTED phrase will act like an alphanumeric *src-item* of the specified size whose value is identical to the characters located on the screen where the DISPLAY will occur. This can be used to modify the video attributes of the screen without changing the displayed data. For example,

   ```
   DISPLAY OMITTED, SIZE 5, REVERSE
   ```

   will cause the five characters located at the current cursor location to be changed to reverse-video.

5. The DISPLAY statement can be used with the "-Cv" IBM DOS/VS COBOL compatibility command line. See DISPLAY UPON SYSPUNCH in section 5.2 of *Transitioning to ACUCOBOL-GT.*

DISPLAY screen-name

Format 2

DISPLAY screen-name displays a Screen Section item defining one or more fields to the screen.

```
DISPLAY screen-name

 [ UPON new-window ]
```

Remaining phrases are optional, can appear in any order.

```
AT screen-loc

AT LINE NUMBER line-num
```

```
AT {COLUMN  } NUMBER col-num
   {COL     }
   {POSITION}
   {POS     }
```

Syntax Rules

1. Different formats of the DISPLAY statement may be mixed together in one DISPLAY statement, as long as no ambiguity results. The effect is the same as specifying each DISPLAY statement separately.
2. *Screen-name* is the name of a screen entry declared in the program's Screen Section.
3. *New-window* is a USAGE HANDLE or PIC X(10) data item.
4. *Screen-loc* is an integer data item or literal containing exactly 4 or 6 digits. It may also be a group item of 4 or 6 characters. If a numeric item is used, it must be a non-negative integer.
5. *Line-num* and *col-num* are numeric data items or literals. They may be non-integer values. You can also specify the value with an arithmetic expression.
6. If the UPON phrase is used, it must be the first optional phrase specified.
7. If the AT phrase is specified, neither the LINE nor the COLUMN phrase may be specified.
8. COLUMN, COL, POSITION, and POS are equivalent.

General Rules

1. A Format 2 DISPLAY statement causes all of the output and update fields in *screen-name* to be displayed on the user's screen. *Screen-name* must be a screen item declared in the program's Screen Section. Before the fields are displayed, each field has its corresponding data item moved to it. Any controls described in *screen-name* are either created or updated as appropriate.

2. When a Format 2 DISPLAY statement executes, spaces are moved to each alphabetic, alphanumeric, and alphanumeric edited input field. Zeros are moved to each numeric and numeric edited input field. Input fields are identified by the word “TO” in their Screen Section entry.

3. The AT, LINE, and COLUMN phrases describe the starting location of the screen item. These phrases are described in detail in **section 6.4.9, “Common Screen Options.”** If either the line or column number is missing or zero, line or column number 1 (one) is used.

DISPLAY WINDOW

Format 3

DISPLAY WINDOW creates and displays a subwindow.

```
DISPLAY {SUBWINDOW}
        {WINDOW   }

 [ UPON new-window ]
```

Remaining phrases are optional, can appear in any order.

```
AT screen-loc

AT LINE NUMBER line-num

AT {COLUMN  } NUMBER col-num
   {COL     }
   {POSITION}
   {POS     }

SIZE length

LINES height

{ERASE} SCREEN
{BLANK}

{REVERSE-VIDEO}
{REVERSE      }
{REVERSED     }
```

```
WITH {COLOR } color-val
     {COLOUR}

{FOREGROUND-COLOR } IS fg-color
{FOREGROUND-COLOUR}

{BACKGROUND-COLOR } IS bg-color
{BACKGROUND-COLOUR}

{HIGHLIGHT}
{HIGH     }
{BOLD     }
{LOWLIGHT }
{LOW      }
{STANDARD }

{BACKGROUND-HIGH    }
{BACKGROUND-LOW     }
{BACKGROUND-STANDARD}

BOXED

SHADOW

[TOP   ] [CENTERED] TITLE IS title
[BOTTOM] [LEFT    ]
         [RIGHT   ]

WITH NO SCROLL

WITH NO WRAP

CONTROL VALUE IS control-val

POP-UP AREA IS save-area
```

Syntax Rules

1. Different formats of the DISPLAY statement may be mixed together in one DISPLAY statement, as long as no ambiguity results. The effect is the same as specifying each DISPLAY statement separately.

2. *New-window* is a USAGE HANDLE or PIC X(10) data item. If used, the UPON phrase must be the first optional phrase specified.

3. *Screen-loc* is an integer data item or literal containing exactly 4 or 6 digits. It may also be a group item of 4 or 6 characters. If a numeric item is used, it must be a non-negative integer.

4. *Line-num*, *col-num*, *length*, and *height* are numeric data items or literals. They may be non-integer values. You can also specify the value of any of these items with an arithmetic expression.

5. *Color-val* is a numeric data item or literal. It can also be an arithmetic expression, except when used in the Screen Section.

6. *Fg-color* and *bg-color* are integer literals or numeric data items. They may be arithmetic expressions. See **section 6.4.9**, "FOREGROUND-COLOR and BACKGROUND-COLOR Phrases," for a more detailed discussion of color settings and values.

7. *Title* is an alphanumeric literal or data item.

8. *Control-value* is a numeric expression.

9. *Save-area* is a USAGE HANDLE or PIC X(10) data item.

10. If the UPON phrase is specified, it must be the first optional phrase.

11. If the AT phrase is specified, neither the LINE nor the COLUMN phrase may be specified.

12. If the COLOR phrase is specified, neither the FOREGROUND-COLOR nor the BACKGROUND-COLOR phrase may be specified.

13. The POP-UP phrase may be specified anywhere in the statement after the required initial elements.

14. WINDOW and SUBWINDOW are synonymous. The SUBWINDOW synonym is available to improve code clarity. Its use makes it clear that a SUBWINDOW is created.

15. IS and "=" are synonymous.

16. COLUMN, COL, POSITION, and POS are equivalent.

17. BLANK and ERASE are equivalent.

18. HIGHLIGHT, HIGH, and BOLD are synonymous.

19. LOWLIGHT and LOW are equivalent.
20. UNDERLINE and UNDERLINED are equivalent.
21. REVERSE-VIDEO, REVERSE, and REVERSED are equivalent.
22. COLOR and COLOUR are synonymous.
23. FOREGROUND-COLOR and FOREGROUND-COLOUR are synonymous.
24. BACKGROUND-COLOR and BACKGROUND-COLOUR are synonymous.
25. The LINES phrase can take a numeric expression.

General Rules

1. The DISPLAY SUBWINDOW verb creates or modifies the current *subwindow.* The subwindow is a rectangular region of the screen. In essence, the current subwindow defines a virtual terminal screen that occupies some area of the user's physical screen. Line and column numbers for ACCEPT and DISPLAY statements are computed from the upper left-hand corner of the current subwindow. For example, the statement DISPLAY SPACE, ERASE SCREEN erases only the current subwindow.
2. When used with floating windows (Format 11), this verb creates a subwindow in the current floating window. Note that every floating window has an implicit subwindow. Each time a floating window is made current, its subwindow is also made current.
3. When used with floating windows, subwindow coordinates are relative to the current floating window. Initially, this is the main application window. The subwindow never extends past the boundaries of the current floating window.
4. The initial subwindow is set to the entire screen. When created inside a floating window, the subwindow is set to the floating window's display area (the area inside the borders, menu bar, toolbar, and title bar).
5. Any subwindows contained in a floating window are automatically closed if the floating window is closed.

LINE NUMBER Phrase

1. The LINE NUMBER phrase sets the top line of the subwindow. In this context, line number one refers to the top line of the current floating window.

2. If this phrase is missing, then the top line of the current floating window is used.

COLUMN NUMBER Phrase

1. The COLUMN NUMBER phrase sets the leftmost column of the window. Column number one refers to the left side of the current floating window.

2. If this phrase is not specified, column number one is used.

AT Phrase

The AT phrase sets both the starting line and column number. It is described in **section 6.4.9, "Common Screen Options."** The line and column numbers arrived at are interpreted as described in the LINE NUMBER and COLUMN NUMBER sections described above.

SIZE Phrase

1. The SIZE phrase sets the number of columns the subwindow will contain. If this causes the window to extend past the right edge of the current floating window, the subwindow's width is adjusted to fit.

2. If this phrase is missing, the subwindow extends to the right edge of the current floating window.

LINES Phrase

1. The LINES phrase sets the number of rows the subwindow will contain. If this causes the window to extend past the bottom of the current floating window, the height will be adjusted to fit.

2. If this phrase is missing, the subwindow extends to the bottom edge of the current floating window.

3. The LINES phrase of a DISPLAY WINDOW, DISPLAY LINE, or DISPLAY BOX verb can take a numeric expression.

ERASE Phrase

1. When the ERASE phrase is specified, the window will be cleared immediately after it is created. Otherwise the window's contents will not be changed. When a window is cleared, it is set to spaces with the current foreground and background colors.

2. This phrase is implied by the BOXED and REVERSED phrases.

BOXED Phrase

1. The BOXED phrase causes a box to be drawn around the new window. The box is drawn *outside* of the window. Any portions of the box that lie off the screen will not be drawn.

2. The terminal's line drawing set is used to draw the box. If the terminal does not have a line drawing set, hyphens and vertical bar characters are used.

3. If the POP-UP phrase is also specified, the box will overlay any other boxes on the screen. If this phrase is not specified, the box drawn will be attached to any other boxes it intersects.

4. This phrase implies the ERASE phrase.

REVERSED Phrase

1. The REVERSED phrase exchanges the window's foreground and background colors. This will affect every ACCEPT and DISPLAY statement in the new window.

2. This phrase implies the ERASE phrase. Note that this will usually cause the entire window to be set to reverse video spaces when it is initially created.

COLOR Phrase

1. The COLOR phrase sets the window's foreground and background colors. These colors are used whenever the window is erased and as default colors for future ACCEPT and DISPLAY statements. *Color-val*

contains a numeric representation of the colors to use. See **section 6.4.9, "Common Screen Options"** COLOR Phrase, for detailed information on setting numeric values for colors.

2. If the foreground color is not specified, the new window inherits the current window's foreground color. If the background color is not specified, the new window inherits the current window's background color.

3. You may use the **WINDOW_INTENSITY** configuration variable to control whether the intensity information in the COLOR setting should be used or ignored (see Appendix H for details).

HIGH, LOW, and STANDARD Phrases

The HIGH, LOW, and STANDARD phrases set the foreground intensity of the subwindow. This affects any drawing done to the subwindow itself (such as its border or title). In addition, they set the default intensity for any future ACCEPT/DISPLAY statements made to this window.

The STANDARD option indicates that a system-dependent default value should be used. You can affect this value with the FOREGROUND-INTENSITY configuration option.

If no option is given, the *current* subwindow's foreground intensity is used (inherit the foreground intensity of the parent).

BACKGROUND-HIGH, BACKGROUND-LOW, and BACKGROUND-STANDARD Phrases

The BACKGROUND-HIGH, BACKGROUND-LOW, and BACKGROUND-STANDARD phrases set the background intensity for the subwindow. This works in a fashion analogous to the foreground intensity described above. Note that the COLOR phrase, if present, takes precedence over the BACKGROUND phrases.

CONTROL VALUE Phrase

The CONTROL VALUE phrase provides a method for specifying certain window characteristics at run time. *Control-value* must be a numeric expression that contains one or more of the following values added together:

Boxed	1
Shadow	2
No Scroll	4
No Wrap	8
Reverse	16

TITLE Phrase

1. The TITLE phrase causes a title to be printed in the window's border. This has effect only if the BOXED phrase is also specified.

2. One top title and one bottom title may be specified for each window. Top titles can be placed in one of three positions in the border region: top left, top center, or top right. Bottom titles can be placed in the bottom left, bottom center, or bottom right. If TOP or BOTTOM is not specified, TOP is used. If LEFT, CENTERED, or RIGHT is not specified, CENTERED is used.

NO SCROLL and NO WRAP Phrases

1. Specifying NO SCROLL disables automatic scrolling for the new window. Normally, when the cursor is moved past the bottom edge of the window, the window is scrolled up one line. If NO SCROLL is specified, then the window will not be scrolled. The bottom line will be overwritten instead. When NO SCROLL is specified, then the only way to scroll a window is explicitly with a SCROLL phrase on a Format 1 DISPLAY statement.

2. Specifying NO WRAP disables line wrap for the window. Normally, a line that extends past the right edge of the window is wrapped around to the next line. If NO WRAP is specified, then the line is truncated instead. This will leave the cursor logically positioned on the same line just to the right of the window's edge. Further output will not be visible until the cursor is repositioned inside the window.

3. The scroll and wrap states of the current window are saved when a pop-up window is created. When that pop-up window is closed, the scroll and wrap states of the old window are restored.

4. Note that the ACUCOBOL-GT runtime system contains two configuration variables, **SCROLL** and **WRAP**, that control the scroll and wrap states of all windows. When these variables are set to zero, then scrolling or wrapping for all windows is disabled regardless of the scroll and wrap states of the individual windows. When these variables are non-zero, then the window's individual states determine the use of scrolling and wrapping.

POP-UP AREA Phrase

1. The POP-UP AREA phrase causes the screen manager to save information about the current subwindow prior to creating the new window. This information can be used by the screen manager later to remove the new window and restore the saved window. This is meant to be used to create "pop-up" windows.

2. Boxed pop-up windows are automatically detached slightly from any intersecting line segments, such as the borders of other windows.

3. The *save-area* is an elementary data item described by a PICTURE X(10) clause. It is filled in with information about the current window dimensions and contents before the new window is created. This data item is required for restoration of a window and must not be subsequently modified in any way. It can be referenced in a CLOSE WINDOW verb to restore the saved window to the screen and re-establish the saved window as the current window.

4. Pop-up subwindows are an older technology that is not compatible with graphical controls. You should avoid using pop-up windows if you also use controls. Use floating windows instead.

SHADOW Phrase

The SHADOW phrase causes the window to appear to float over the screen, giving it a three-dimensional effect.

The way the shadow is displayed is determined by the SHADOW-STYLE setting of the SCREEN option in your runtime configuration file.

When a shadow is specified for a window, that window is automatically detached slightly from any intersecting line segments, such as the borders of other windows.

Note: Phrases not described above are described in **section 6.4.9, "Common Screen Options."**

DISPLAY SCREEN SIZE

Format 4

DISPLAY SCREEN SIZE shifts the display between 80 and 132 column mode

```
DISPLAY SCREEN SIZE  {80 }
                     {132}

   [ ON EXCEPTION statement-1 ]

   [ NOT ON EXCEPTION statement-2 ]

   [ END-DISPLAY ]
```

Syntax Rules

1. Different formats of the DISPLAY statement may be mixed together in one DISPLAY statement, as long as no ambiguity results. The effect is the same as specifying each DISPLAY statement separately.

2. *Statement-1* and *statement-2* are imperative statements.

General Rules

1. The DISPLAY SCREEN verb is used to shift between the 80-column mode and 132-column mode of the terminal. On character-based systems, the terminal is physically set to either 132- or 80-column mode. For all systems, the main application window is then set to 132 columns wide; it is cleared, and its subwindow set to cover the entire interior. Some graphical systems simulate 132-column mode by scrolling the main application window.

In graphical environments, the MODIFY verb is the preferred method for changing the size of a graphical screen.

2. If the terminal hardware does not support the mode shifted to, the screen and current window do not change and the EXCEPTION phrase *statement-1* (if specified) executes. If the mode change is successful and the NOT EXCEPTION phrase is used, *statement-2* executes.

DISPLAY LINE

Format 5

DISPLAY LINE draws a horizontal or vertical line on the screen.

```
DISPLAY LINE

 [ UPON new-window ]

   { SIZE length  }
   { LINES height }
```

Remaining phrases are optional, can appear in any order.

```
AT screen-loc

AT LINE NUMBER line-num

AT {COLUMN  } NUMBER col-num
   {COL     }
   {POSITION}
   {POS     }

{REVERSE-VIDEO}
{REVERSE      }
{REVERSED     }

WITH {COLOR } color-val
     {COLOUR}

[CENTERED] TITLE IS title
[LEFT    ]
[RIGHT   ]
```

Syntax Rules

1. Different formats of the DISPLAY statement may be mixed together in one DISPLAY statement, as long as no ambiguity results. The effect is the same as specifying each DISPLAY statement separately.
2. *New-window* is a USAGE HANDLE or PIC X(10) data item. If used, the UPON phrase must be the first optional phrase specified.
3. *Screen-loc* is an integer data item or literal containing exactly 4 or 6 digits. It may also be a group item of 4 or 6 characters. If a numeric item is used, it must be a non-negative integer.
4. *Line-num*, *col-num*, *length*, and *height* are numeric data items or literals. They may be non-integer values. You can also specify the value of any of these items with an arithmetic expression.
5. *Color-val* is a numeric data item, literal, or arithmetic expression.
6. *Title* is an alphanumeric literal or data item.
7. If the UPON phrase is specified, it must be the first optional phrase.
8. If the AT phrase is specified, neither the LINE nor the COLUMN phrase may be specified.
9. Exactly one of the SIZE or LINES phrases must be specified. The selected phrase may appear anywhere in the statement.
10. The LINES phrase can take a numeric expression.
11. IS and "=" are synonymous.
12. COLUMN, COL, POSITION, and POS are equivalent.
13. REVERSE-VIDEO, REVERSE, and REVERSED are equivalent.
14. COLOR and COLOUR are synonymous.

General Rules

1. The DISPLAY LINE verb provides the ability to draw horizontal and vertical lines in a machine- and terminal-independent fashion. The lines are drawn using the best mode available on the display device. Used together with the DISPLAY BOX verb, this verb provides the ability to draw forms on the user's screen.

   DISPLAY LINE is an older, character-based technology. If you want fine control over lines displayed on graphical systems, use the BAR control instead. See chapter 5, Book 2, *User Interface Programming.*

2. The appropriate intersection character (corner or three-way intersection) is used when drawn lines intersect other lines on the screen.

3. If the SIZE phrase is specified, the line drawn is horizontal. The value of *length* gives the size of the line in screen columns. If the LINES phrase is used instead, the line drawn is a vertical line and *height* describes the number of screen rows to use.

4. Lines never wrap around or cause scrolling. If the LINES or SIZE phrase would cause the line to leave the current window, the line is truncated at the edge of the window.

AT, LINE, and COLUMN Phrases

1. The value of *line-num* gives the starting row of the line. The value of *col-num* gives the starting column. The value of *screen-loc* gives the starting row and column. (For details, see **section 6.4.9, "Common Screen Options."**) Lines are always drawn to the right or downwards as appropriate. *Screen-loc*, *line-num*, and *col-num* must specify a position that is contained in the current window.

2. If the LINE NUMBER phrase is not specified, line one is used. If the COLUMN NUMBER phrase is missing, column one is used.

TITLE Phrase

1. The TITLE phrase has effect only when you are drawing horizontal lines. When it is specified, *title-string* is printed in part of the line.

2. The title may be printed near the right side, near the left side, or in the center of the line depending whether the RIGHT, LEFT, or CENTERED phrase is specified. If none is specified, CENTERED is used.

COLOR and REVERSE Phrases

The COLOR and REVERSE phrases operate in the same manner as described in **section 6.4.9, "Common Screen Options."** Note, however, that the COLOR value may not specify blinking, underlining, or an intensity. Lines are always drawn in a terminal-dependent intensity.

Note: Phrases not described above are described in **section 6.4.9, "Common Screen Options."**

DISPLAY BOX

Format 6

DISPLAY BOX draws a box on the screen.

```
DISPLAY BOX

 [ UPON new-window ]
```

Remaining phrases are optional, can appear in any order.

```
AT screen-loc

AT LINE NUMBER line-num

AT {COLUMN  } NUMBER col-num
   {COL     }
   {POSITION}
   {POS     }

SIZE length

LINES height

{REVERSE-VIDEO}
```

```
{REVERSE      }
{REVERSED     }

WITH {COLOR } color-val
     {COLOUR}

[TOP   ] [CENTERED] TITLE IS title
[BOTTOM] [LEFT    ]
         [RIGHT   ]
```

Syntax Rules

1. Different formats of the DISPLAY statement may be mixed together in one DISPLAY statement, as long as no ambiguity results. The effect is the same as specifying each DISPLAY statement separately.
2. *New-window* is a USAGE HANDLE or PIC X(10) data item. If used, the UPON phrase must be the first optional phrase specified.
3. *Screen-loc* is an integer data item or literal containing exactly 4 or 6 digits. It may also be a group item of 4 or 6 characters. If a numeric item is used, it must be a non-negative integer.
4. *Line-num*, *col-num*, *length*, and *height* are numeric data items or literals. They may be non-integer values, but only the integer value will be applied. You can also specify the value of any of these items with an arithmetic expression.
5. *Color-val* is a numeric data item, literal, or arithmetic expression.
6. *Title* is an alphanumeric literal or data item.
7. If the UPON phrase is specified, it must be the first optional phrase.
8. If the AT phrase is specified, neither the LINE nor the COLUMN phrase may be specified.
9. The LINES phrase can take a numeric expression.
10. IS and “=” are synonymous.
11. COLUMN, COL, POSITION, and POS are equivalent.
12. REVERSE-VIDEO, REVERSE, and REVERSED are equivalent.
13. COLOR and COLOUR are synonymous.

General Rules

1. The DISPLAY BOX verb provides the ability to draw a box in a machine- and terminal-independent manner. The best drawing mode of the display device is used. If the lines used in drawing a box intersect other lines already present on the screen, the appropriate intersection characters are used.

2. DISPLAY BOX is an older, character-based technology. For fine control of boxes on graphical systems, use the FRAME control. See Chapter 5, Book 2, *User Interface Programming*.

3. You specify the location of the box by providing the location of the upper-left corner. You specify the size of the box by providing a height and a width.

AT, LINE, and COLUMN Phrases

The AT, LINE NUMBER, and COLUMN NUMBER phrases operate in the same manner as they do when they are used in a DISPLAY LINE statement (Format 5).

SIZE and LINES Phrases

The SIZE phrase specifies the width of the box. The LINES phrase specifies its height. *Length* and *height* must specify values greater than one. If the SIZE phrase is absent, the box will extend to the right edge of the current window. If the LINES phrase is missing, the box will extend to the bottom of the current window.

COLOR and REVERSE Phrases

The COLOR and REVERSE phrases operate in the same manner as described in **section 6.4.9, "Common Screen Options."** Note that the COLOR value may not specify blinking, underlining, or an intensity (boxes are always drawn in a terminal-dependent intensity).

TITLE Phrase

The TITLE phrase operates in the same manner as it does for a DISPLAY WINDOW verb (Format 3).

Note: Phrases not described above are described in **section 6.4.9, "Common Screen Options."**

DISPLAY UPON WINDOW TITLE

Format 7

DISPLAY UPON WINDOW TITLE modifies a subwindow's title

```
DISPLAY source UPON WINDOW [TOP   ] [CENTERED] TITLE
                           [BOTTOM] [LEFT    ]
                                    [RIGHT   ]
```

Syntax Rules

1. Different formats of the DISPLAY statement may be mixed together in one DISPLAY statement, as long as no ambiguity results. The effect is the same as specifying each DISPLAY statement separately.
2. *Source* is an alphanumeric data item or nonnumeric literal.

General Rules

1. The DISPLAY UPON WINDOW TITLE verb is used to modify a boxed subwindow's title. Either the top or bottom title may be modified.
2. If any of the positioning phrases is used, the new title is placed in the indicated position.
3. If neither the TOP/BOTTOM nor the CENTERED/LEFT/RIGHT option is used, then the window's title is modified in its current position. If the window has both a top and bottom title, the top title is modified.
4. If the TOP/BOTTOM phrase is omitted, TOP is implied.
5. If the CENTERED/LEFT/RIGHT phrase is not used, CENTERED is implied.

DISPLAY UPON COMMAND LINE

Format 8

DISPLAY UPON COMMAND LINE changes the contents of the command-line buffer

```
DISPLAY cmd-line UPON COMMAND-LINE
```

Syntax Rules

1. Different formats of the DISPLAY statement may be mixed together in one DISPLAY statement, as long as no ambiguity results. The effect is the same as specifying each DISPLAY statement separately.
2. *Cmd-line* is an alphanumeric data item or nonnumeric literal.

General Rules

1. The DISPLAY UPON COMMAND-LINE verb is used to move the contents of an alphanumeric data item into the buffer where the command line is stored. The command line is *not* re-executed. The only effect of this verb is that it changes the value that will be returned by subsequent ACCEPT FROM COMMAND-LINE statements.
2. You can also access the command-line buffer from a C program. The buffer is an external data array named Acmd_line.

DISPLAY src-item (ANSI format)

Format 9

DISPLAY src-item *(ANSI format)* sends data directly to the screen without using the ACUCOBOL-GT window manager.

```
DISPLAY {src-item} ... [ UPON mnemonic-name ] [ WITH NO ADVANCING ]
```

Syntax Rules

1. Different formats of the DISPLAY statement may be mixed together in one DISPLAY statement, as long as no ambiguity results. The effect is the same as specifying each DISPLAY statement separately.

2. *Src-item* is a literal or data item that specifies the data to be displayed. It must be USAGE DISPLAY unless the CONVERSION phrase or the "-Vd" compiler option is specified. Note that by default the screen is cleared when the program starts. If you specify the "-Ca" compiler option, the screen is not cleared until the first DISPLAY statement with the CONVERSION phrase is executed. If you want the conversion to take place without the screen being cleared, you must specify the "-Ca" and "-Vd" options without the CONVERSION phrase on the DISPLAY statement.

3. *Mnemonic-name* must be a user-defined word declared in Special-Names that refers to a display device, or it must be the name of the display device itself. See **section 4.2.3, "Special-Names Paragraph,"** for a list of valid devices.

4. If the UPON phrase is specified it must be the first optional phrase.

5. If the UPON phrase is omitted, then the "-Ca" compiler option must be specified. If it is not, then the statement is treated as a Format 1 DISPLAY statement instead.

General Rules

1. The data contained in *src-item* is sent to the output device named by the UPON phrase. If the UPON phrase is omitted, the output device is assumed to be the user's console.

2. If the WITH NO ADVANCING phrase is omitted, then the device is advanced to the next line after the last *src-item* is displayed. If WITH NO ADVANCING is specified, no additional action takes place after the data is sent to the output device. If line advancing is used, trailing spaces are removed from the last *src-item* before it is sent to the device.

3. If the output device is any of the following, the data is sent to the user's console:

   ```
   CONSOLE, SYSOUT, SYSLST, SYSLIST
   ```

This data may be redirected with the "-o" runtime option or with operating system commands. If line advancing is used, the system's output buffer is flushed, but if NO ADVANCING is specified, then the buffer is not flushed. When data is sent to these devices, no editing or formatting of the data occurs.

4. If the output device is SYSOUT-FLUSH, the data is sent to the user's console. After the last *src-item* is sent, the output buffer is flushed, forcing the data to be sent immediately. No editing or formatting of the data occurs.

5. If the output device is SYSERR, the data is sent to the runtime's error output. This is either the user's screen or the file specified in the "-e" runtime option.

6. Technical Note: This form of the DISPLAY statement sends data directly to the user's screen without using ACUCOBOL-GT's window manager. On many machines, the window manager maintains an image of the user's screen in memory. (This improves efficiency by omitting redundant screen displays and is used to implement "pop-up" windows.) By sending data directly to the screen, you can cause the window manager's screen image to be in error. This can cause strange effects when mixed with ACUCOBOL-GT-specific DISPLAY statements, including:

   - lost data
   - incorrect functioning of CLOSE WINDOW
   - incorrect cursor position
   - incorrect character attributes
   - incorrect display in debugger

   For these reasons, you must be careful when using ANSI DISPLAY. Here are some useful guidelines:

   - If your DISPLAY does not affect the screen image, then it's OK to use it. For example, you can use ANSI DISPLAY to send a control sequence to an attached printer or cash register.

- If you use only ANSI DISPLAY statements, then you should not experience any problems (except that the debugger will not be able to restore the user's screen). With this approach, you should avoid all DISPLAY statements except for Format 9.

- If you must mix formats, then you can use the library routine "W$FORGET" to correct the behavior of the window manager. This routine causes the window manager to enter its initial state, where it does not know the screen image or current attribute settings. If you call this routine after a series of Format 9 DISPLAY statements, the window manager will be set to a state where it can correctly manage the screen. Note that this routine will cause the cursor to be positioned to the last line on the screen.

- You can always use UPON SYSERR safely. When sent to the screen, the output is directed through the window manager. Normally, you would use this with the "-e" runtime option to direct error messages to an error log file.

- In the DOS/VS COBOL compatibility mode, output may be sent to a simulated card punch. See "DISPLAY UPON SYSPUNCH" in section 5.2 of *Transitioning to ACUCOBOL-GT.*

DISPLAY UPON GLOBAL TITLE

Format 10

DISPLAY UPON GLOBAL TITLE changes the title of the application window in a graphical user interface, or the title of a floating window.

```
DISPLAY title-1 UPON { FLOATING WINDOW handle-1 } TITLE
                     { GLOBAL WINDOW            }
```

Syntax Rules

1. Different formats of the DISPLAY statement may be mixed together in one DISPLAY statement, as long as no ambiguity results. The effect is the same as specifying each DISPLAY statement separately.

2. *Title-1* is an alphanumeric literal or identifier that contains the new title.

3. *Handle-1* is the handle of the floating window in which the new title is applied.
4. If the UPON phrase is specified, it must be the first optional phrase.

General Rules

1. DISPLAY UPON FLOATING WINDOW TITLE is used to change the title of the main application window or floating window. When you are changing a floating window's title, *handle-1* identifies which window to change.
2. The case of the title will be exactly as given in *title-1*.
3. Alternatively, you can use the MODIFY verb to change a window's title.

DISPLAY FLOATING WINDOW

Format 11

DISPLAY FLOATING WINDOW creates and displays a floating window.

```
DISPLAY FLOATING [GRAPHICAL] WINDOW

 [ UPON parent-window ]
```

Remaining phrases are optional, can appear in any order.

```
{MODELESS}
{MODAL   }

{LINK} TO THREAD
{BIND}

SCREEN LINE NUMBER screen-line

SCREEN {COLUMN  } NUMBER screen-col
       {COL     }
       {POSITION}
       {POS     }

AT screen-loc
```

```
AT LINE NUMBER line-num

AT {COLUMN  } NUMBER col-num
   {COL     }
   {POSITION}
   {POS     }

SIZE  length

LINES height

FONT {IS} font-1
     {= }

CONTROL FONT {IS} font-3
             {= }

CELL
 {SIZE  }  [IS] {cell-units                                  }
 {HEIGHT}  [= ] {control-type-name  FONT font-2 [SEPARATE  ]}
 {WIDTH }       {control-type-name  FONT        [OVERLAPPED]}

{ERASE} SCREEN
{BLANK}

{REVERSE-VIDEO}
{REVERSE      }
{REVERSED     }

WITH {COLOR } color-val
     {COLOUR}

{FOREGROUND-COLOR } IS fg-color
{FOREGROUND-COLOUR}

{BACKGROUND-COLOR } IS bg-color
{BACKGROUND-COLOUR}

{HIGHLIGHT}
{HIGH     }
{BOLD     }
{LOWLIGHT }
{LOW      }
{STANDARD }
```

```
{BACKGROUND-HIGH    }
{BACKGROUND-LOW     }
{BACKGROUND-STANDARD}

{ [USER-GRAY] [USER-WHITE] }
{        USER-COLORS       }

BOXED

SHADOW

TITLE-BAR

[TOP   ] [CENTERED] TITLE IS title
[BOTTOM] [LEFT    ]
         [RIGHT   ]

WITH SYSTEM MENU

WITH NO SCROLL

WITH NO WRAP

{NO-CLOSE}

{AUTO-RESIZE}

{RESIZABLE  }

MIN-SIZE  {= } min-size
          {IS}
MAX-SIZE  {= } max-size
          {IS}
MIN-LINES {= } min-lines
          {IS}
MAX-LINES {= } max-lines
          {IS}

CONTROL VALUE {IS} control-val
              {= }

LAYOUT-MANAGER {IS} manager
               {= }
```

```
VISIBLE {IS} {TRUE         }
        {= } {FALSE        }
             {visible-state}

POP-UP MENU {IS}  {menu-1}
            {= }  {NULL}

{POP-UP AREA IS } handle-name
{HANDLE {IS}    }
        {IN}

CONTROLS-UNCROPPED

EVENT PROCEDURE IS { proc-1 [ {THROUGH} proc-2 ] }
                             {THRU   }
                   { NULL                       }

ACTION {IS} action
       {= }
```

Syntax Rules

1. Different formats of the DISPLAY statement may be mixed together in one DISPLAY statement, as long as no ambiguity results. The effect is the same as specifying each DISPLAY statement separately.

2. *Parent-window* is a USAGE HANDLE or PIC X(10) data item. If used, the UPON phrase must be the first optional phrase specified.

3. *Screen-line* and *screen-col* are numeric expressions.

4. *Screen-loc* is an integer data item or literal containing exactly 4 or 6 digits. It may also be a group item of 4 or 6 characters. If a numeric item is used, it must be a non-negative integer.

5. *Line-num*, *col-num*, *length*, and *height* are numeric data items or literals. They may be non-integer values. You can also specify the value of any of these items with an arithmetic expression.

6. *Font-1, font-2* and *font-3* are data items described as USAGE HANDLE or HANDLE OF FONT. They should contain valid handles to screen fonts.

7. *Cell-units* is a positive integer data item or literal.

8. *Control-type-name* is one of the control type reserved words known by the compiler.

9. *Color-val* is a numeric data item or literal. *Color-val* can also be an arithmetic expression, except when used in the Screen Section.

10. *Fg-color* and *bg-color* are integer literals or numeric data items. They may be arithmetic expressions. See **section 6.4.9**, "FOREGROUND-COLOR and BACKGROUND-COLOR Phrases", for a more detailed discussion of color settings and values.

11. *Min-size, max-size, min-lines* and *max-lines* are integer literals or data items.

12. *Title* is an alphanumeric literal or data item.

13. *Control-value* is a numeric expression.

14. The word "NO-CLOSE" is reserved by the compiler only when it appears in a Format 11 or 12 DISPLAY statement.

15. *Manager* is a USAGE HANDLE or HANDLE OF LAYOUT-MANAGER that contains a valid reference to a layout manager.

16. *Visible-state* is a numeric literal or data item.

17. *Menu-1* is a USAGE HANDLE or HANDLE OF MENU data item.

18. *Handle-name* is a USAGE HANDLE, HANDLE OF WINDOW, or PIC X(10) data item.

19. *Proc-1* and *proc-2* are procedure names.

20. *Action* is a numeric literal or data item.

21. You must compile allowing for recursive paragraphs in order to specify the EVENT PROCEDURE phrase. Compiling for recursive paragraphs is allowed by default, but can be turned off with the "-Zr0" compiler option.

22. If the UPON phrase is specified, it must be the first optional phrase.

23. The SCREEN LINE phrase and the SCREEN COLUMN phrase must be used together. If they are used, you cannot use the AT, LINE, or COLUMN phrases.

24. If the AT phrase is specified, neither the LINE nor the COLUMN phrase may be specified.
25. If the COLOR phrase is specified, neither the FOREGROUND-COLOR nor the BACKGROUND-COLOR phrase may be specified.
26. The POP-UP/HANDLE phrase may be specified anywhere in the statement after the required initial elements.
27. IS and "=" are synonymous.
28. COLUMN, COL, POSITION, and POS are equivalent.
29. BLANK and ERASE are equivalent.
30. HIGHLIGHT, HIGH, and BOLD are synonymous.
31. LOWLIGHT and LOW are equivalent.
32. REVERSE-VIDEO, REVERSE, and REVERSED are equivalent.
33. COLOR and COLOUR are synonymous.
34. FOREGROUND-COLOR and FOREGROUND-COLOUR are synonymous.
35. BACKGROUND-COLOR and BACKGROUND-COLOUR are synonymous.

General Rules

1. The syntax for DISPLAY FLOATING WINDOW is a superset of the DISPLAY WINDOW verb. This simplifies conversion of DISPLAY WINDOW statements to DISPLAY FLOATING WINDOW statements.
2. The DISPLAY FLOATING WINDOW verb creates a new floating window and stores a handle to the window in *handle-name*. Use the value of *handle-name* with other verbs (such as DESTROY) when you need to refer to the window.
3. After the new window is created, it becomes both the current and active window.

4. The window created may be either *modal* or *modeless*. A modal window is a window that the user cannot leave until it is closed. A modeless window is a window that the user can leave (switch to another window) while it is still open. These names are derived from the idea that a modal window enters a new *mode* in the program (for example, selecting a file to open) while a modeless window does not (since the user can continue working on tasks in other windows).

5. The DISPLAY FLOATING WINDOW verb also creates a new subwindow that exactly covers the interior of the floating window. This is identical to an implied DISPLAY SUBWINDOW statement (with no options). Any HIGH, LOW, STANDARD, BACKGROUND-HIGH, BACKGROUND-LOW, BACKGROUND-STANDARD, REVERSE, COLOR, NO SCROLL, or NO WRAP phrases specified in the DISPLAY FLOATING WINDOW verb are inherited by the implied subwindow.

6. Each window has a *controlling* thread. A window's controlling thread is the most recent thread to have created that window or done an ACCEPT from that window. When a thread performs an ACCEPT from a window, and that thread is not the controlling thread of the active window, the thread suspends. The thread remains suspended until the window it is accessing becomes active (either because the user activates it or the program does).

7. Most of the optional phrases have the same meaning as they do for DISPLAY SUBWINDOW. However, note the following exceptions:

    a. The ERASE SCREEN phrase is always implied by DISPLAY FLOATING WINDOW, so specifying this phrase has no additional effect.

    b. Most GUIs (including Windows) cannot display shadowed pop-up windows. On these systems, the SHADOW phrase has no effect. On character-based systems, the SHADOW phrase has its normal effect.

    c. All GUIs create borders around their windows. If there is a choice of border thickness, specifying BOXED will select a thicker border than omitting BOXED. Under character-based systems, the BOXED phrase determines whether or not there will be a border.

d. The HIGH, LOW, STANDARD, BACKGROUND-HIGH, BACKGROUND-LOW, BACKGROUND-STANDARD, REVERSE, COLOR, NO WRAP, and NO SCROLL phrases do not directly affect the created window. Instead, they are passed on to the initial subwindow as described in Rule 5, above.

e. Most GUIs (including Windows) cannot display more than one window title and do not give you a choice of title position. On these systems, the specified TITLE appears in the location determined by the GUI (usually top center, or top left). If you specify more than one title, the TOP title is the one used. If you specify only one title (either TOP or BOTTOM), it is used regardless of the title location.

GRAPHICAL Phrase

The optional GRAPHICAL phrase directs the compiler to use a default CELL phrase equivalent to:

```
CELL SIZE = LABEL FONT
```

This phrase establishes the window's coordinate space based on the font used by controls that occupy the window. The CELL phrase can still be used and any values set in that phrase take precedence over the default value established with GRAPHICAL option. In other words, if you specify only a CELL HEIGHT or CELL WIDTH, then the other dimension receives the default assignment.

The intent of the GRAPHICAL option is to make it easier to consistently establish an appropriate coordinate space for windows that contain only controls (see the discussion of cell sizing and coordinate space that is included with the CELL phrase rules, below).

For example, the window that is specified with:

```
DISPLAY FLOATING WINDOW,
    CELL SIZE = LABEL FONT
```

can be more simply specified with:

```
DISPLAY FLOATING GRAPHICAL WINDOW
```

UPON Phrase

The UPON phrase specifies the parent of the new floating window. *Parent-window* must be a valid floating window handle. If the UPON phrase is omitted, the current window is used as the parent. If you create a new floating window in the scope of an UPON phrase, the new window becomes the current window when the DISPLAY statement terminates.

MODAL and MODELESS Phrases

1. The word MODAL makes a floating window modal. Floating windows are modal by default, so this word is just commentary. When a modal window is active, all other windows are disabled. The user cannot activate another window, including any of its components (such as its menu or close button). Note, however, that while a user cannot activate another window, the program can (see Format 10 of the SET statement).

2. The word MODELESS makes a window modeless. When a modeless window is active, the user can activate another window by using the host system's techniques for doing so (for example, by clicking on another window with the mouse). When this happens, any ACCEPT that is active is terminated by a CMD-ACTIVATE event. Your program should respond by performing an ACCEPT in the window requested by the user. Alternatively, you can link your modeless window to a thread, see rule 1 under LINK TO THREAD and BIND TO THREAD Phrases (below).

LINK TO THREAD and BIND TO THREAD Phrases

1. The LINK TO THREAD phrase allows the runtime to automate the handling of the CMD-ACTIVATE event. If the user activates a window created with the LINK TO THREAD phrase, the runtime will examine that window to see if it has a controlling thread different from the current thread. If it does, then the current thread suspends and the thread controlling the newly active window is allowed to run. The runtime handles all aspects of the window activation. The CMD-ACTIVATE event is not returned to the program in this case. If the controlling thread of the new window is the same as the current thread, then the runtime does not perform any special handling and the CMD-ACTIVATE event is passed on to your program. In order to get the best benefit from the LINK TO THREAD phrase, you should arrange to have a separate thread control each modeless window in your program.

2. The BIND TO THREAD phrase has the same effect as the LINK TO THREAD phrase. In addition, the window is automatically destroyed when its controlling thread terminates.

SCREEN LINE and SCREEN COLUMN Phrases

The SCREEN LINE and SCREEN COLUMN phrases determine the initial location of the window on the screen. *Screen-line* and *screen-col* give the coordinates of the upper left corner of the window in screen base units. Screen base units are machine dependent. On character systems, they are character cells. On graphical systems, they are pixels. The upper left corner of the screen is location "1,1". Under Windows, the runtime ensures that the initial window is fully visible, so the specified location may not be used if that would place a portion of the window off the screen (the closest allowed location is used). Windows other than the initial window may be placed arbitrarily. On graphical systems, the location of a floating window is interpreted to mean the location of its exterior. On character systems, the location is the same as it is for subwindows: the location of the window's interior.

LINE, COLUMN, and AT Phrases

1. The LINE phrase indicates the starting row of the new window. This is always relative to the first line of the parent window. Non-integer values are allowed. If the LINE phrase is omitted, then the new window is first centered vertically over the parent window and then adjusted to be fully on the screen.

   Note: This rule differs from the handling of DISPLAY SUBWINDOW. With DISPLAY SUBWINDOW, omitting the LINE phrase is the same as specifying "LINE 1".

   In all cases, the positioning is relative to the parent window as physically displayed on the screen, ignoring any aspect of the window not currently displayed. Thus, if the current window is the main application window, and that window has been scrolled by the user, line "1" refers to the first line of the physical window--not the (undisplayed) first line of the main application window.

2. The COLUMN phrase works analogously to the LINE phrase, except that it controls the horizontal positioning.

3. The AT phrase *screen-loc* item must be either a 4-digit or 6-digit number. The first half of this number is the starting row, the second half the starting column. These values are interpreted in the same manner as they are for the LINE and COLUMN phrases. A value of zero is treated as zero (i.e., `AT 0000` is equivalent to `ROW 0, COL 0`).

SIZE and LINES Phrases

1. The SIZE phrase indicates the width of the interior of the new window. If it is omitted, then the width is the same as the main application window. If there is no main application window available, the default size of the floating window is the same as the current window.

Note: A non-integer SIZE is allowed. If the SIZE is not an integer, then the partial column created cannot be used to display textual characters; however, graphical controls can be located there. The partial column is always shown as spaces with the floating window's background color. The minimum SIZE value allowed is "1". The runtime currently limits the maximum size to 132 columns.

---

2. The LINES phrase indicates the height of the new window's interior. If it is omitted, then the height is the same as the main application window. As with the SIZE phrase, a non-integer number of lines may be specified. Any partial lines created are always displayed as spaces with the background color. The minimum value for LINES is "1" (one). The runtime currently limits the maximum size to 50 lines.

FONT Phrase

The FONT phrase assigns the font that will be used for all textual ACCEPT and DISPLAY statements used in the window. This also sets the default *cell size* to the size of the "0" (zero) character described by *font-1*. The cell size determines the height of one row and the width of one column. The font described by *font-1* must be a fixed-width font. If it is not, or if the FONT phrase is not specified, then the font used is the same as the one used by the parent window.

CONTROL FONT Phrase

The CONTROL FONT phrase specifies the default font to use for any graphical controls displayed in this window. If you omit the CONTROL FONT phrase, a system default is used (the font "DEFAULT-FONT").

CELL Phrase

The CELL phrase defines the height, or width, or height and width of one cell in the window. A cell defines the height of one row and the width of one column. The default cell size is set by the size of the font used in the window.

The cell size is described in terms of cell units. The exact meaning of a cell unit is machine-dependent. Typically, for character-based systems, one cell unit is equal to the height or width (as appropriate) of one screen character. On graphical systems, a cell unit is typically one pixel in size. When developing programs, you should avoid writing code that depends on fixed (hard-coded) values for cell units.

The HEIGHT option of the CELL phrase defines the cell height for the new window. The WIDTH option defines the width. The SIZE phrase defines both the height and width together.

The cell-units option sets the cell's height, or width, or both, to the value of cell-units.

The control-type-name phrase causes the cell height, or width, or both, to be based on a particular font and control type. The system measures the size of *font-2* when it is used in a control described by control-type-name, and sets the cell size accordingly. This option is typically used to set the coordinate space of the window to one that is convenient for aligning several controls of a particular type and font. Note that the font handle (*font-2*) is not required. When it is omitted, the window's CONTROL FONT is used. Also note that if the font handle is omitted, the optional word FONT is required (in order to avoid ambiguity with the FONT phrase).

If the SEPARATE option is specified, then a system-dependent amount is added to the measured font height to provide for some vertical separation between controls. This is typically used to provide some space between

boxed entry-fields on adjacent rows. On the other hand, if OVERLAPPED is specified, the height is reduced by the size of the top border of a boxed entry field. This causes boxed fields on adjacent rows to share a common border.

The runtime currently limits control-type-name to be either a “LABEL” or “ENTRY-FIELD”. If another control type name is used, the runtime treats it as if it were type “LABEL”.

If a window’s cell width does not match the width of its font, or if its cell height is less than the height of its font, then the effects of a textual ACCEPT or DISPLAY statement in that window are undefined. If its cell height is larger than its font’s height, then the characters are positioned at the top of each cell, and the lower portion of the cell is filled with the text’s background color.

Note: The purpose of the CELL phrase is to simplify the construction of windows that contain only controls. We strongly recommend that you use it (or the GRAPHICAL phrase described above) whenever you create a window that will only contain controls. If you do not, your screens will be less portable.

In particular, if the relative size of the font you use in your controls changes in relation to the system’s fixed font, then you will experience problems, including overlapping controls. This is because the default cell size that defines the coordinate space is based on a fixed-size font in order to maintain compatibility with character-based applications. The size relationship between the variable-pitch font used in controls and the default fixed-font that defines the coordinate space determines the appearance of the screen. If the relationship changes, the appearance of the screen changes. One way that this can happen is if the end user’s machine is missing one of the fonts. In this case, Windows will substitute a different font, which may be a different size. To avoid these problems, define your coordinate space based on the same font that your controls use (with the CELL phrase). Then if the font changes the entire screen is rescaled uniformly.

For example, the following statement defines the coordinate space based on the font used with entry fields. This definition allows you to easily position entry fields vertically with “LINE 1”, “LINE 2”, etc., and have it look right.

```
CELL SIZE IS ENTRY-FIELD FONT SEPARATE
```

USER-GRAY, USER-WHITE, and USER-COLORS Phrases

The USER-GRAY, USER-WHITE, and USER-COLORS phrases provide a convenient way of matching your application's normal colors to those chosen by the user. The USER-GRAY option causes the palette manager to map color number "8" (low-intensity white) to the color that the user has chosen to use with 3-D objects on the host system. Similarly, USER-WHITE maps color number "16" (high-intensity white) to the color the user has chosen to be the normal background color for application windows. If you arrange your application so that it uses color number "8" as the background for regions populated with graphical controls, and color number "16" for plain text regions, your application will look much like other applications on the system.

The USER-COLORS phrase indicates that you want to apply both the USER-GRAY and USER-WHITE options. These phrases are effective only on host graphical systems that have a palette manager. On other systems, these phrases have no effect. Also, note that the palette applies to the entire application. Because of this, you usually specify these options only on the first window you create.

Note: Because Windows make abundant use of 3-D effects in displaying controls, we strongly suggest that you use the USER-GRAY or USER-COLORS options for programs with graphical controls that run under Windows. Graphical controls look best when placed on a "gray" background (color number "8"). Other color choices may make 3-D controls look odd.

TITLE-BAR Phrase

The TITLE-BAR phrase indicates that you want to have a title bar placed along the top edge of the new window. This phrase is automatically implied by the TITLE phrase (exception: this is not true if you also use the CONTROL VALUE phrase). Under some GUIs (including Windows), you must place a title bar in order to move the floating window with the mouse. Without a title bar, the user's ability to move the window depends on the host GUI. Note that you can have a title bar without specifying a title.

SYSTEM MENU Phrase

The SYSTEM MENU phrase causes a system menu (also known as a close box) to appear on the created window. This menu allows the user to close the floating window. It may also have additional properties depending on the host system. Under Windows, this menu contains the Move and Close operations. If you include a system menu, your program must be ready to act on a close window event (cmd_close) at any time. See **section 4.2**, and Chapter 6, Book 2, *User Interface Programming*, for additional information. See also the **QUIT_MODE** configuration variable in Appendix H, Book 4, for shutdown handling options. Note that the SYSTEM MENU phrase also implies the TITLE-BAR phrase.

NO-CLOSE Phrase

The NO-CLOSE phrase causes the window's "Close" menu option to be disabled. This option can be applied only when the window is created and its effects cannot be reversed (the associated window's "Close" option is permanently disabled). The NO-CLOSE option takes precedence over other settings, including the setting of the **QUIT_MODE** configuration variable.

AUTO-RESIZE and RESIZABLE Phrases

1. The AUTO-RESIZE phrase specifies that the window be displayed with *resizable* borders. When the window is created, it is displayed full size as defined by the SIZE and LINES phrases. By dragging the resizable borders the user can reduce or increase the size of the window. The runtime automatically adds scroll bars as needed and manages any required scrolling. The window also has a maximize button that allows the user to immediately resize the window to its full size. The exact representation and functioning of the resizable borders and the maximize button is host system dependent. Although the user can change the physical size of the window, the logical size does not change. Neither do controls in the window change size or position. If AUTO-RESIZE is omitted, the window is a fixed size.

2. The RESIZABLE phrase creates a window that the user can resize but omits the automatic handling provided by the AUTO-RESIZE phrase. When the user resizes the window, the size of the logical window is changed to match the new physical window. Any area that is new is displayed with spaces in the window's background color. Any area that has been removed is lost (although any permanent controls in that area will still exist). The window's subwindow is resized to fill the

interior of the resized window. The subwindow's background color is changed to match the window's background color. Other traits of the subwindow remain unchanged. The program receives a NTF-RESIZED event to inform it of the new size. See the section on events for details. Windows that have the RESIZABLE attribute can use a resize layout manager to help handle the resizing and positioning of controls in the window. See section 4.8.4, "The Resize Layout Manager," in Book 2 for details.

3. For windows with the RESIZABLE phrase, *min-size* and *min-lines* set the windows' smallest width and height respectively. This value is expressed in character cells (fractional cells are ignored). If omitted, or set to zero, the smallest window size is determined by the host system. Similarly, the *max-size* and *max-lines* values set the window's largest width and height. If omitted, or set to zero, the host system determines the largest size (usually the entire screen). For windows without the RESIZABLE phrase, these values are ignored.

Note: Adding or removing a menu or toolbar from a window normally causes the window to be resized to maintain its interior dimensions. For windows with the RESIZABLE phrase, the window is not resized. Instead, the interior dimensions are reduced or increased as needed. You should either modify the window to be the desired size, or inquire the current dimensions when you add or remove a menu or toolbar for resizable windows.

ACTION Phrase

The ACTION phrase allows you to programmatically maximize, minimize, or restore a window. To use ACTION, assign it one of the following values (these names are found in acugui.def):

ACTION-MAXIMIZE	maximizes the window. It has the same effect as if the user clicked the "maximize" button. Allowed only for windows that have RESIZABLE or AUTO-RESIZE specified or implied for them.
ACTION-MINIMIZE	minimizes the window. Allowed only with INDEPENDENT windows that have the AUTO-MINIMIZE property set to true. It is not supported with other types of floating windows; if set, it is ignored by the runtime. ACTION-MINIMIZE has the same effect as if the user clicked the "minimize" button.
ACTION-RESTORE	If the window is currently maximized or minimized, restores the window to its previous size and position; otherwise, it has no effect. Allowed only for windows that can be maximized or minimized.

4. If you assign an ACTION value that is not allowed, then there is no effect other than to trigger the ON EXCEPTION phrase of the MODIFY statement (if present). Note that you can use the ACTION phrase to create a window that is initially maximized or minimized.

CONTROL VALUE Phrase

The CONTROL VALUE phrase allows you to specify certain attributes of the new window at run time instead of at compile time. *Control-val* must be a numeric expression. In it, you can specify certain floating window traits by adding together any of the following values:

Boxed	1
Shadow	2
No Scroll	4
No Wrap	8
Reverse	16
Title-Bar	32
System Menu	64
User-Gray	128
User-White	256

For each value specified, the corresponding attribute is given to the new window. When a value is not specified, the presence or absence of that trait depends on the other phrases included in the DISPLAY FLOATING WINDOW statement. Note that you can only give traits to a window with the CONTROL VALUE phrase; you cannot negate traits specified by the DISPLAY FLOATING WINDOW statement. For example, if you want to specify at run time whether or not a window gets a shadow, you should omit the SHADOW phrase from the DISPLAY FLOATING WINDOW statement and use a CONTROL VALUE phrase to add shadowing when you want it.

LAYOUT-MANAGER Phrase

The LAYOUT-MANAGER phrase attaches a layout manager to the window.

VISIBLE Phrase

The VISIBLE option determines whether the window created is visible or invisible. If the FALSE option is used, or *visible-state* is the value zero, then the window is invisible. Otherwise, the window is visible. If the VISIBLE phrase is omitted, then the window is visible.

POP-UP MENU Phrase

The POP-UP MENU phrase associates a pop-up menu with the window. If *menu-1* is specified, then the menu associated with *menu-1* becomes the pop-up menu. If NULL is specified, the window is not given a pop-up menu. Pop-up menus are activated by a machine-dependent technique. Under Windows, the technique is to right-click on the window's background.

CONTROLS-UNCROPPED Phrase

Normally, when you create a control in a window, the control is cropped to fit the current subwindow's dimensions. In addition, if the control's home position is outside of the current subwindow, the control is not created. Adding the phrase CONTROLS-UNCROPPED overrides these rules. When this phrase is used, the control is created with the specified location and dimensions, regardless of whether the control will be physically in the window.

This can be useful when you are dealing with RESIZABLE windows. Sometimes a resizable window is too small to show all of the controls that your program creates. Normally, these controls either would not be created or would be cropped. This could produce odd results when the window is later resized larger by the user. Although the resized window is now large enough to show everything, the controls still show their cropped appearance, because their (cropped) creation size is recorded in the controls as their actual size. Specifying CONTROLS-UNCROPPED avoids the cropping behavior.

This style is useful also when you want to place a combo-box near the bottom of a window. Because the size of the drop-down portion of the combo-box is determined by the control's overall height, cropping the control limits the drop-down box to the window's boundaries. If you want the box to drop down beyond the edge of the window, you need to use the CONTROLS-UNCROPPED window style to allow this.

EVENT PROCEDURE Phrase

1. A window's event procedure is executed whenever an event is processed for that window. The event procedure is executed as if it were the target of a PERFORM statement. Only the window's own events trigger the event procedure. Events generated by controls contained in the window do not trigger the window's event procedure (they trigger the control's

event procedure instead). The event procedure executes while the event is being processed, before the event causes termination of any executing ACCEPT statement. See **section 5.9.6, "PROCEDURE Clause,"** for more information about event procedures.

2. Specifying *proc-1* assigns that procedure as the window's event procedure. Flow of control returns at the end of *proc-1*, unless *proc-2* is specified, in which case flow of control returns at the end of *proc-2*. If you specify the NULL option, the window does not have an event procedure. This is the default, so the NULL option is treated as commentary.

Note: Phrases not described above are described in **section 6.4.9, "Common Screen Options."**

DISPLAY INITIAL WINDOW

Format 12

DISPLAY INITIAL WINDOW creates and displays the main application window and independent windows.

```
DISPLAY {INITIAL    } [GRAPHICAL] WINDOW
        {STANDARD   }
        {INDEPENDENT}
```

Remaining phrases are optional, can appear in any order.

```
MODELESS

LINK TO THREAD

SCREEN LINE NUMBER screen-line

SCREEN {COLUMN  } NUMBER screen-col
       {COL     }
       {POSITION}
       {POS     }

AT screen-loc                          (independent only)
```

```
AT LINE NUMBER screen-line          (independent only)

AT {COLUMN  } NUMBER screen-col     (independent only)
   {COL     }
   {POSITION}
   {POS     }

SIZE  length

LINES height

FONT {IS} font-1
     {= }

CONTROL FONT {IS} font-3
             {= }

CELL
 {SIZE  } [IS]  {cell-units                                 }
 {HEIGHT} [= ]  {control-type-name  FONT font-2 [SEPARATE  ]}
 {WIDTH }       {control-type-name  FONT        [OVERLAPPED]}

{ERASE} SCREEN
{BLANK}

{REVERSE-VIDEO}
{REVERSE      }
{REVERSED     }

WITH {COLOR } color-val
     {COLOUR}

{FOREGROUND-COLOR } IS fg-color
{FOREGROUND-COLOUR}

{BACKGROUND-COLOR } IS bg-color
{BACKGROUND-COLOUR}

{HIGHLIGHT}
{HIGH     }
{BOLD     }
{LOWLIGHT }
{LOW      }
{STANDARD }
```

```
{BACKGROUND-HIGH    }
{BACKGROUND-LOW     }
{BACKGROUND-STANDARD}

{ [USER-GRAY] [USER-WHITE] }
{        USER-COLORS       }

TITLE-BAR

[TOP   ] [CENTERED] TITLE IS title
[BOTTOM] [LEFT    ]
         [RIGHT   ]

WITH SYSTEM MENU

WITH NO SCROLL

WITH NO WRAP

{NO-CLOSE}

{AUTO-RESIZE}
{RESIZABLE  }

MIN-SIZE  {= } min-size
          {IS}
MAX-SIZE  {= } max-size
          {IS}
MIN-LINES {= } min-lines
          {IS}
MAX-LINES {= } max-lines
          {IS}

AUTO-MINIMIZE

CONTROL VALUE {IS} control-val
              {= }

LAYOUT-MANAGER {IS} manager
               {= }

VISIBLE {IS} {TRUE         }
        {= } {FALSE        }
             {visible-state}
```

```
POP-UP MENU {IS}  {menu-1}
            {= }  {NULL}

{POP-UP AREA IS  } handle-name
{HANDLE     {IS} }
            {IN}

CONTROLS-UNCROPPED

EVENT PROCEDURE IS { proc-1 [ {THROUGH} proc-2 ] }
                               {THRU   }
                   { NULL                       }
```

Syntax Rules

1. Different formats of the DISPLAY statement may be mixed together in one DISPLAY statement, as long as no ambiguity results. The effect is the same as specifying each DISPLAY statement separately.

2. *Screen-line* and *screen-col* are numeric expressions.

3. *Screen-loc* is an integer data item or literal containing exactly 4 or 6 digits. It may also be a group item of 4 or 6 characters. If a numeric item is used, it must be a non-negative integer.

4. *Line-num*, *col-num*, *length*, and *height* are numeric data items or literals. They may be non-integer values. You can also specify the value of any of these items with an arithmetic expression.

5. *Font-1*, *font-2* and *font-3* are data items described as USAGE HANDLE or HANDLE OF FONT. They should contain valid handles to screen fonts.

6. *Cell-units* is a positive integer data item or literal.

7. *Control-type-name* is one of the control type reserved words known by the compiler.

8. *Color-val* is a numeric data item or literal. *Color-val* can also be an arithmetic expression, except when used in the Screen Section.

9. *Fg-color* and *bg-color* are integer literals or numeric data items. They may be arithmetic expressions. See **section 6.4.9**, “FOREGROUND-COLOR and BACKGROUND-COLOR Phrases,” for a more detailed discussion of color settings and values.

10. *Title* is an alphanumeric literal or data item.
11. *Min-size*, *max-size*, *min-lines* and *max-lines* are integer literals or data items.
12. *Control-value* is a numeric expression.
13. The word “NO-CLOSE” is reserved by the compiler only when it appears in a Format 11 or 12 DISPLAY statement.
14. *Manager* is a USAGE HANDLE or HANDLE OF LAYOUT-MANAGER that contains a valid reference to a layout manager.
15. *Visible-state* is a numeric literal or data item.
16. *Menu-1* is a USAGE HANDLE or HANDLE OF MENU data item.
17. *Handle-name* is a USAGE HANDLE, HANDLE OF WINDOW, or PIC X(10) data item.
18. *Proc-1* and *proc-2* are procedure names.
19. You must compile allowing for recursive paragraphs in order to specify the EVENT PROCEDURE phrase. Compiling for recursive paragraphs is allowed by default, but can be turned off with the “-Zr0” compiler option.
20. The SCREEN LINE phrase and the SCREEN COLUMN phrase must be used together. If they are used, you cannot use the AT, LINE, or COLUMN phrases.
21. If the COLOR phrase is specified, neither the FOREGROUND-COLOR nor the BACKGROUND-COLOR phrase may be specified.
22. The POP-UP/HANDLE phrase may be specified anywhere in the statement after the required initial elements.
23. IS and “=” are synonymous.
24. COLUMN, COL, POSITION, and POS are equivalent.
25. BLANK and ERASE are equivalent.
26. HIGHLIGHT, HIGH, and BOLD are synonymous.

27. LOWLIGHT and LOW are equivalent.
28. REVERSE-VIDEO, REVERSE, and REVERSED are equivalent.
29. COLOR and COLOUR are synonymous.
30. FOREGROUND-COLOR and FOREGROUND-COLOUR are synonymous.
31. BACKGROUND-COLOR and BACKGROUND-COLOUR are synonymous.

General Rules

1. The DISPLAY INITIAL WINDOW verb creates the main application window. The main application window has several special properties. If it is minimized, all other windows in the application are also minimized. If it is closed, the application terminates. A program can have only one main application window.
2. If you attempt to create a main application window after one already exists, the DISPLAY INITIAL WINDOW statement will have no effect other than to set *handle-name* to NULL.
3. The runtime automatically constructs the main application window if needed. This occurs any time a screen operation is dictated by the program and the program has not yet constructed a main application window. When this occurs, the runtime executes the following implied statement:

```
DISPLAY INITIAL WINDOW
    TITLE-BAR,
    SYSTEM MENU,
    AUTO-MINIMIZE,
    AUTO-RESIZE.
```

4. The main application window is always *modeless*. A modeless window is one where the user can switch to another window while the current window is still open. You can include the word MODELESS in the statement as commentary.
5. The INDEPENDENT phrase creates an independent window. Independent windows act like additional main application windows. Independent windows have the following traits:

a. Independent windows do not have a parent. As a result, any other window in the application can be placed over them. Also, destroying another window in the application will not destroy the independent window.

b. Although they do not have a parent, independent windows use the *current* window to determine their default fonts, cell size, and colors. Also, independent windows use the current window when determining their position. This is computed in the same manner as it is for floating windows.

c. Independent windows can be minimized separately. Under Windows and Windows NT, each visible independent window has its own button on the task bar.

d. Independent windows process their close box in the same manner as floating windows--by generating a CMD-CLOSE event.

e. Independent windows can be created before the main window. In this case, there is no current window to provide defaults, so the independent window uses the same defaults as the main application window would. The window is located on the screen as follows:

   According to SCREEN LINE and SCREEN COL, if specified; otherwise

   If the window has a title bar, the host system places the window as if it were a new application; otherwise

   The window is centered in the screen.

   Note that LINE and COL (without the SCREEN option) are ignored in these cases.

f. If an independent window is current when the main application window is created, the defaults for the main window are derived from the independent window.

6. Most of the phrases allowed for DISPLAY INITIAL WINDOW work in exactly the same way that they work in a DISPLAY FLOATING WINDOW (format 11) statement. The following rules are supplemental.

SCREEN-LINE and SCREEN-COLUMN Phrases

Under Windows, the runtime ensures that the initial window is fully visible, so the location specified by *screen-line* and *screen-col* may not be used if that would place a portion of the window off the screen (the closest allowed location is used).

TITLE-BAR Phrase

If you specify a TITLE-BAR but do not give a TITLE, the default title is the name of the program (PROGRAM-ID in the IDENTIFICATION DIVISION).

AUTO-MINIMIZE Phrase

1. The AUTO-MINIMIZE phrase indicates that a minimize button should be displayed. The runtime handles the minimizing and restoring of the application automatically. If you do not specify AUTO-MINIMIZE, the user is not allowed to minimize the application.

2. In addition, the AUTO-MINIMIZE phrase implies the SYSTEM MENU and TITLE-BAR phrases.

Window location and size

1. The window's location on the screen is determined by the host system, as are its initial dimensions if it is resizable. You can use the **The SCREEN Option** runtime configuration variable to specify the initial window dimensions (including whether it is maximized or minimized).

   On a character-based system, the initial window always occupies the entire terminal surface. Setting the various sizing options has no effect.

2. If you do not use the LINES and SIZE phrases, the lines and size values are taken from the SIZE option of the SCREEN runtime configuration variable. If SCREEN is not set, the default size is 25 lines by 80 columns. The host system may provide a smaller default (for example, 24-line character-based terminals will default to 24 lines).

STANDARD Phrase

The STANDARD option is identical to the INITIAL option except that it automatically implies the following options:

1. TITLE-BAR
2. SYSTEM MENU
3. AUTO-MINIMIZE
4. USER-COLORS
5. For graphical systems, a black foreground on a white background. For character-based systems, a white foreground on a black background. You may override these default colors with the various color setting phrases.

GRAPHICAL Phrase

The GRAPHICAL phrase has the same effect as in the DISPLAY FLOATING WINDOW statement.

VISIBLE Phrase

The VISIBLE option determines whether the window created is visible or invisible. If the FALSE option is used, or *visible-state* is the value zero, then the window is invisible. Otherwise, the window is visible. If the VISIBLE phrase is omitted, then the window is visible.

POP-UP MENU Phrase

The POP-UP MENU phrase associates a pop-up menu with the window. If *menu-1* is specified, then the menu associated with *menu-1* becomes the pop-up menu. If NULL is specified, the window is not given a pop-up menu. Pop-up menus are activated by a machine-dependent technique. Under Windows, the technique is to right-click on the window's background.

CONTROLS-UNCROPPED Phrase

Normally, when you create a control in a window, the control is cropped to fit the current subwindow's dimensions. In addition, if the control's home position is outside of the current subwindow, the control is not created. Adding the phrase CONTROLS-UNCROPPED overrides these rules. When

this phrase is used, the control is created with the specified location and dimensions, regardless of whether the control will be physically in the window.

This can be useful when you are dealing with RESIZABLE windows. Sometimes a resizable window is too small to show all of the controls that your program creates. Normally, these controls either would not be created or would be cropped. This could produce odd results when the window is later resized larger by the user. Although the resized window is now large enough to show everything, the controls still show their cropped appearance, because their (cropped) creation size is recorded in the controls as their actual size. Specifying CONTROLS-UNCROPPED avoids the cropping behavior.

This style is useful also when you want to place a combo-box near the bottom of a window. Because the size of the drop-down portion of the combo-box is determined by the control's overall height, cropping the control limits the drop-down box to the window's boundaries. If you want the box to drop down beyond the edge of the window, you need to use the CONTROLS-UNCROPPED window style to allow this.

EVENT PROCEDURE Phrase

1. A window's event procedure is executed whenever an event is processed for that window. The event procedure is executed as if it were the target of a PERFORM statement. Only the window's own events trigger the event procedure. Events generated by controls contained in the window do not trigger the window's event procedure (they trigger the control's event procedure instead). The event procedure executes while the event is being processed, before the event causes termination of any executing ACCEPT statement. See **section 5.9.6, "PROCEDURE Clause,"** for more information about event procedures.

2. Specifying *proc-1* assigns that procedure as the window's event procedure. Flow of control returns at the end of *proc-1*, unless *proc-2* is specified, in which case flow of control returns at the end of *proc-2*. If you specify the NULL option, the window does not have an event procedure. This is the default, so the NULL option is treated as commentary.

Tip: The Support area of the Micro Focus Web site includes a sample program that demonstrates how a main application window can be coded to detect the system display's screen resolution and then size itself to fill the entire screen. To download the program, go to: **http:// supportline.microfocus.com/xmlloader.asp?type=home.** Select **Examples and Utilites** > **Acucorp examples** > **Graphical User Interface Sample Programs** > **WIN-START-MAX.cbl.**

DISPLAY TOOL-BAR

Format 13

DISPLAY TOOL-BAR adds a toolbar to the current window.

```
DISPLAY TOOL-BAR

HANDLE {IS} handle-name
       {IN}
```

Remaining phrases are optional, can appear in any order.

```
LINES {IS} height [CELL ]
      {= }        [CELLS]

CONTROL FONT {IS} font-1
             {= }

CELL
 {SIZE  }  [IS] {cell-units                                  }
 {HEIGHT}  [= ] {control-type-name  FONT font-2 [SEPARATE  ]}
 {WIDTH }       {control-type-name  FONT        [OVERLAPPED]}

WITH {COLOR } color-val
     {COLOUR}

{FOREGROUND-COLOR } IS fg-color
{FOREGROUND-COLOUR}

{BACKGROUND-COLOR } IS bg-color
{BACKGROUND-COLOUR}
```

```
{HIGHLIGHT}
{HIGH     }
{BOLD     }
{LOWLIGHT }
{LOW      }
{STANDARD }

{BACKGROUND-HIGH    }
{BACKGROUND-LOW     }
{BACKGROUND-STANDARD}
```

Syntax Rules

1. Different formats of the DISPLAY statement may be mixed together in one DISPLAY statement, as long as no ambiguity results. The effect is the same as specifying each DISPLAY statement separately.

2. *Height* is a numeric data item or literal. It may be a non-integer value. You can also specify the value as an arithmetic expression.

3. *Color-val* is a numeric data item, numeric literal, or arithmetic expression

4. *Fg-color* and *bg-color* are integer literals or numeric data items. They may be arithmetic expressions. See **section 6.4.9**, "FOREGROUND-COLOR and BACKGROUND-COLOR Phrases", for a more detailed discussion of color settings and values.

5. *Font-1* is a USAGE HANDLE or HANDLE OF FONT data item. It should contain a valid handle to a screen font.

6. *Handle-name* is a USAGE HANDLE, HANDLE OF WINDOW, or PIC X(10) data item.

7. If the COLOR phrase is specified, neither the FOREGROUND-COLOR nor the BACKGROUND-COLOR phrase may be specified.

8. The HANDLE phrase may be specified anywhere in the statement after the required initial elements.

9. IS and "=" are synonymous.

10. HIGHLIGHT, HIGH and BOLD are synonymous.

11. LOWLIGHT and LOW are equivalent.
12. COLOR and COLOUR are synonymous.
13. FOREGROUND-COLOR and FOREGROUND-COLOUR are synonymous.
14. BACKGROUND-COLOR and BACKGROUND-COLOUR are synonymous.
15. *Cell-units* is a positive integer data item or literal.

General Rules

1. DISPLAY TOOL-BAR adds a toolbar to the current floating or main-application window. A toolbar is a region of the window that is devoted to holding buttons and other controls that are used as *mouse accelerators* that the user can activate with the mouse to quickly specify a program operation. Toolbars extend across the entire width of the window. They appear at either the top or the bottom of the window depending on the host system. Under Microsoft Windows, toolbars appear at the top of the window.

   On character-based systems, toolbars are created but are always invisible. This is done to simplify cross-platform support: you can unconditionally create toolbars in your application. They will seem to behave properly on character-based systems, but they will not actually show up.

2. You may have more than one toolbar associated with a particular window. All toolbars associated with a window are stacked vertically in the order that they are created. You may have a maximum of five toolbars associated with each window.

3. Like menu-bars, the space that a toolbar occupies is *not* part of the body (*client area*) of the window. When you create a toolbar, the window it is attached to is enlarged to accommodate the space required by the toolbar (unless the window is RESIZABLE; see the description of the RESIZABLE phrase in the General Rules section of the Format 11 DISPLAY statement). You do not lose any application space when adding a toolbar to a window, except as constrained by the physical screen. Toolbars do not scroll when the body of the window is

scrolled. Instead, the toolbar remains accessible at the top of the window. Toolbars automatically grow and shrink horizontally to match the width of the owning window.

4. In several respects, toolbars act like windows. Because of this, they are called *child windows* of the window they are attached to. When you create a toolbar, a handle that identifies it is returned in *handle-name*. Like other window handles, you can use *handle-name* in an UPON phrase of a DISPLAY statement to direct output to the toolbar. You can also use *handle-name* in a DESTROY statement to remove the toolbar. Unlike other windows, toolbars are not made current or active when they are created. This means that the only way to place a control on the toolbar is with the UPON phrase.

5. Toolbars cannot take textual output (i.e., text-style ACCEPT and DISPLAY statements cannot refer to a toolbar). You can only place graphical controls on a toolbar. However, any type of graphical control may be placed on a toolbar. Most commonly, push buttons appear on toolbars (either text or bitmap buttons), as well as bitmap check-boxes and radio-buttons.

6. *Font-1* identifies the default font to use for any controls shown in the toolbar. If *font-1* is not specified, DEFAULT-FONT is used.

7. *Height* specifies the height of the toolbar. Height units are derived from the height of *font-1* (or DEFAULT-FONT if *font-1* is not used). If *height* is omitted, then the toolbar uses a host-dependent default height.

8. The color and intensity phrases have their usual meaning as described in **section 6.4.9, "Common Screen Options."** The toolbar directly uses the background-color and background-intensity only when it is drawn. The foreground-color and intensity are used as defaults for controls displayed on the toolbar (see the relevant sections of Chapter 5, Book 2, *ACUCOBOL-GT User Interface Programming*, for information about how a particular control type uses the default foreground color and intensity). Any color and intensity elements that are omitted from the DISPLAY TOOL-BAR statement are inherited from the window that the toolbar is attached to.

9. **Programmer's Note:** The easiest way to make effective use of a toolbar is to populate it with controls that can act like function keys. When you do this, the toolbar acts in a manner that is very similar to a

menu bar. This simplifies other programming because you do not need to attempt to directly activate the controls on the toolbar. Controls that can act like function keys are push buttons, check boxes and radio buttons. For push buttons, use the SELF-ACT style. For check boxes and radio buttons, use both the SELF-ACT and NOTIFY styles. This ensures that these controls behave like function keys (i.e., they activate themselves when clicked, and inform your program).

DISPLAY control-type-name

Format 14

DISPLAY control-type-name creates a graphical control, including a graphical ActiveX or COM control.

```
DISPLAY {control-type-name  }
        {OBJECT control-type}

 [ title ]

 [ UPON new-window ]
```

Remaining phrases are optional, can appear in any order.

```
{IDENTIFICATION} {IS} control-id
{ID            } {= }

AT screen-loc   [CELL  ]
                [CELLS ]
                [PIXEL ]
                [PIXELS]

AT LINE NUMBER line-num   [CELL  ]
                          [CELLS ]
                          [PIXEL ]
                          [PIXELS]

AT {COLUMN  } NUMBER col-num   [CELL  ]
   {COL     }                  [CELLS ]
   {POSITION}                  [PIXEL ]
   {POS     }                  [PIXELS]

AT CLINE NUMBER cline-num   [CELL ]
```

```
                                [CELLS]

AT CCOL NUMBER ccol-num    [CELL ]
                           [CELLS]

SIZE  {IS} length [CELL  ]
      {= }        [CELLS ]
                  [PIXEL ]
                  [PIXELS]

LINES {IS} height [CELL  ]
      {= }        [CELLS ]
                  [PIXEL ]
                  [PIXELS]

CSIZE   {IS} clength  [CELL ]
        {= }          [CELLS]

CLINES  {IS} cheight  [CELL ]
        {= }          [CELLS]

MAX-HEIGHT {IS} max-height
           {= }

MAX-WIDTH  {IS} max-width
           {= }

MIN-HEIGHT {IS} min-height
           {= }

MIN-WIDTH  {IS} min-width
           {= }

TITLE {IS} title
      {= }

KEY   {IS} key-letter
      {= }

{COLOR } IS color-val
{COLOUR}

{FOREGROUND-COLOR } IS fg-color
{FOREGROUND-COLOUR}
```

```
{BACKGROUND-COLOR } IS bg-color
{BACKGROUND-COLOUR}

{HIGHLIGHT}
{HIGH     }
{BOLD     }
{LOWLIGHT }
{LOW      }
{STANDARD }

{BACKGROUND-HIGH    }
{BACKGROUND-LOW     }
{BACKGROUND-STANDARD}

STYLE {IS} style-flags
      {= }

{style-name} ...

VALUE {IS} [ MULTIPLE ] value [ LENGTH {IS} length-1 ]
      {= } [ TABLE    ]               {= }

FONT {IS} font-handle
     {= }

HANDLE {IN} control-handle
       {IS}

LAYOUT-DATA {IS} layout-data
            {= }

ENABLED  {IS} {TRUE         }
         {= } {FALSE        }
              {enabled-state}

VISIBLE  {IS} {TRUE         }
         {= } {FALSE        }
              {visible-state}

POP-UP MENU {IS}  {menu-1}
            {= }  {NULL}

HELP-ID {IS} help-id
        {= }
```

```
EVENT-LIST     {IS} ( event-value { event-value ... } )
               {= }

AX-EVENT-LIST  {IS} ( ax-event-value { ax-event-value ... } )
               {= }

EXCLUDE-EVENT-LIST {IS} list-state
                   {= }

EVENT PROCEDURE IS { proc-1 [ {THROUGH} proc-2 ] }
                             {THRU   }
                   { NULL                       }

{{property-name         } {IS } property-option
                                    [GIVING result-1]}. ...
 {PROPERTY property-type} {ARE}
```

where *property-option* is one of the following:

```
{ property-value [ LENGTH {IS} length-1 ] }
{                         {= }            }
{ ( {property-value} ... )                }
{                                          }
{ {MULTIPLE} property-table                }
{ {TABLE   }                               }
```

Syntax Rules

1. Different formats of the DISPLAY statement may be mixed together in one DISPLAY statement, as long as no ambiguity results. The effect is the same as specifying each DISPLAY statement separately.

2. *Control-type-name* is one of the control type reserved words known by the compiler.

3. *Control-type* is a numeric literal or data item.

4. *New-window* is a USAGE HANDLE or PIC X(10) data items. If used, the UPON phrase must be the first optional phrase specified.

5. *Control-id* is an integer numeric literal or data item. Its value should be in the range of 0 to 32767.

6. *Screen-loc* is an integer data item or literal containing exactly 4, 6, or 8 digits. It may also be a group item of 4, 6, or 8 characters. If a numeric item is used, it must be a non-negative integer.

7. *Line-num, col-num, cline-num, ccol-num, length, height, clength, and cheight* are numeric data items or literals. They may be non-integer values, except when pixels are specified. You can also specify the value of any of these items with an arithmetic expression.

8. *Max-height, max-width, min-height,* and *min-width* are numeric data items or literals.

9. *Color-val* is a numeric data item or literal. *Color-val* can also be an arithmetic expression.

10. *Fg-color* and *bg-color* are integer literals or numeric data items. They may be arithmetic expressions. See **section 6.4.9**, "FOREGROUND-COLOR and BACKGROUND-COLOR Phrases", for a more detailed discussion of color settings and values.

11. *Title* is an alphanumeric literal or data item. In Format 14, *title* may appear only once, either in a TITLE phrase or the initial *title* option. The *title* option must be the first option specified.

12. *Key-letter* is an alphanumeric literal or data item.

13. *Font-handle* is a data item of type USAGE HANDLE or HANDLE OF FONT.

14. *Style-flags* is a numeric expression.

15. *Style-name* is the name of a style associated with the class of control being described. If *control-type-name* is omitted, then you may not use the *style-name* phrase. You may use the STYLE phrase instead.

16. *Value* is a literal or data item. If the MULTIPLE option is specified, then *value* must be a one-dimensional table. In this case, *value* is not subscripted.

17. *Length-1* is a numeric literal or data item. The LENGTH phrase may be specified only if the immediately preceding *value* or *property-value* is an alphanumeric literal or data item, and not a figurative constant. In addition, the MULTIPLE option may not be specified along with the LENGTH phrase.

18. *Control-handle* is a USAGE HANDLE data item.

19. *Layout-data* is an integer literal, data item, or expression.

20. *Enabled-state*, *visible-state*, and *help-id* are integer numeric literals or data items.

21. *Menu-1* is a USAGE HANDLE or HANDLE OF MENU data item.

22. *Event-value* and *ax-event-value* are numeric literals or data items that identify an event type. List elements must be enclosed in parentheses. Elements must be separated by a space. If the list contains a single element, the parentheses can be omitted.

23. *List-state* is an integer literal or numeric data item. Valid values are "0" and "1".

24. *Proc-1* and *proc-2* are procedure names.

25. *Property-name* is the name of a property specific to the type of control being referenced. If the type of control is unknown to the compiler (as in a "DISPLAY OBJECT object-1" statement), then *property-name* may not be used. You must use the PROPERTY *property-type* option instead.

26. *Property-type* is a numeric literal or data item. It identifies the property to modify. The numeric values that identify the various control properties can be found in the COPY library "controls.def".

27. *Property-value* is a literal or data item. *Property-value* may also be a numeric expression (however, only the first *property-value* in a phrase may be an expression, subsequent values must be literals or data items). When multiple values are specified, the parentheses are required. An example of their use is: DISPLAY-COLUMNS = (1, 20)

28. *Property-table* is a data item that appears in a one-dimensional table. No index should be specified.

29. *Result-1* is a numeric data item.

30. If the *title* option is used, it must be the first optional phrase.

31. If the UPON phrase is specified, it must be the first optional phrase, except when the *title* option is used, then the UPON phrase must be second, following the *title* phrase.

32. If the AT phrase is specified, neither the LINE nor the COLUMN phrase may be specified.

33. If the COLOR phrase is specified, neither the FOREGROUND-COLOR nor the BACKGROUND-COLOR phrase may be specified.

34. If the CELLS option is used in either the SIZE or CSIZE phrase, it must be present in both phrases if both are specified. The same rule applies to the CELLS option in the LINES and CLINES phrases.

35. The HANDLE phrase may be specified anywhere in the statement after the required initial elements.

36. IS and "=" are synonymous.

37. COLUMN, COL, POSITION, and POS are equivalent.

38. HIGHLIGHT, HIGH, and BOLD are synonymous.

39. LOWLIGHT and LOW are equivalent.

40. COLOR and COLOUR are synonymous.

41. FOREGROUND-COLOR and FOREGROUND-COLOUR are synonymous.

42. BACKGROUND-COLOR and BACKGROUND-COLOUR are synonymous.

43. MULTIPLE and TABLE are synonymous.

General Rules

1. The DISPLAY control-type-name verb creates a new control and displays it on the screen. A handle to the new control is returned in *control-handle*. The handle is used in future references to the control. If *control-handle* is not specified, the control is an *anonymous control*. Anonymous controls can be referenced in future statements only by position.

2. *Control-type-name* identifies the type of the control. The exact set of controls available and their types are discussed in Chapter 5, Book 2, *User Interface Programming.* Use OBJECT *control-type* when the control type is not known at compile time. At run time, *control-type*

must contain the identifying number of a control type known to the system. If it does not correspond to any control type, the DISPLAY statement is ignored and *control-handle* is set to NULL. The identifying number of each control type is defined in the "controls.def" COPY file.

3. The LENGTH option of the VALUE and property phrases establishes the exact size of the source data to use. Without the LENGTH option, a text value presented to the control is the data contained in the COBOL data element with trailing spaces removed. When you specify the LENGTH option, exactly *length-1* characters of the data are used instead. No trailing space removal occurs. This is useful in a few cases when you need to treat trailing spaces as data.

   If *length-1* is a value larger than the size of the data item it is modifying, then the size of the data item is used instead. If *length-1* is negative, it is ignored and the default handling occurs.

4. Some properties return specific values when set. These values are placed in *result-1* of the GIVING phrase. The meaning of the value depends on the property being set; see the documentation for the specific property. Properties that do not have a pre-defined return value set *result-1* to "1" if the property was set successfully, or "0" if not. When a property is being given multiple values in a single assignment (for example: "Display-Columns = (1, 10, 30)"), then *result-1* is set in response to the last value assigned.

5. After the DISPLAY statement is executed, the cursor is positioned at the first whole column immediately to the right of the control. If this column is past the right edge of the current subwindow, the cursor is not moved from its previous location.

6. Once created, a control remains in existence until one of the following events:

   a. The control is explicitly destroyed with a DESTROY verb; or,

   b. if the control has the TEMPORARY style, the control is "overwritten" (as defined in **section 5.2, "Data Names"**); or

   c. the window containing the control is destroyed.

7. Under some systems, controls do not interact properly with *pop-up* subwindows. This is due to the fact that the controls are maintained by the GUI directly while pop-up subwindows are managed by the ACUCOBOL-GT runtime system. Since the GUI is not aware of the pop-up nature of the subwindow, it displays the control over the subwindow, even if the program's intent is to have the pop-up subwindow on top. For this reason, pop-up subwindows are not recommended for programs that will be using graphical controls. You should use floating windows instead.

8. The POP-UP MENU phrase associates a pop-up menu with the control. If *menu-1* is specified, then the menu associated with *menu-1* becomes the pop-up menu. If NULL is specified, the control is not given a pop-up menu. Pop-up menus are activated by a machine-dependent technique. Under Windows, the technique is to right-click on the control.

9. See **section 6.4.9, "Common Screen Options,"** for a description of the remaining optional phrases. Any phrases not described there behave in the same manner as in a Format 1 DISPLAY statement.

10. The runtime ignores events from all controls while it is creating an ActiveX control, via the DISPLAY statement or otherwise. If you are using a control that delivers significant information using events and you don't want to miss those events while you are creating a new control, set the **CONTROL_CREATION_EVENTS** runtime configuration variable to "1" (On, True, Yes). Alternatively, you could avoid creating an ActiveX control when you are expecting an event.

DISPLAY MESSAGE BOX

Format 15

DISPLAY MESSAGE BOX creates a simple message box

```
DISPLAY MESSAGE BOX

 { text } . . .
```

Remaining phrases are optional and may be used in any order.

```
TITLE {IS} title
      {= }

TYPE {IS} type
     {= }

ICON {IS} icon
     {= }

DEFAULT {IS} default
        {= }

{GIVING   } value
{RETURNING}
```

Syntax Rules

1. *Text* is a literal or data item.
2. *Title* is an alphanumeric literal or data item.
3. *Type*, *icon*, and *default* are numeric literals or data items.
4. *Value* is a numeric data item.

General Rules

1. Format 15 (DISPLAY MESSAGE BOX) creates a simple modal pop-up window with a title-bar, a text message, an icon (on some systems) and one or more push buttons. It then waits for the user to push one of the buttons and returns the results. The window is then destroyed. Message boxes come in "OK" and "Yes/No" formats, with an optional "Cancel" button in each format. Message boxes are a programming convenience when you need to create a simple dialog box that can fit one of these predefined formats.
2. *Text* forms the body of the message box. It is the string of text that the user will see. When more than one *text* item is specified, they are concatenated together to form a single string that the user sees. The limit on the length of the message string is host dependent. The host system wraps the text as needed to fit the message box. If you want to force the text to a new line, you can embed an ASCII line-feed

character (h”0A” in COBOL) where you want the new line to start. For example, the following code produces a message box with two lines of text:

```
78  NEWLINE      VALUE H"0A".

DISPLAY MESSAGE BOX,
   "This is line 1", NEWLINE,

   "and this is line 2" .
```

3. When *text* is numeric, it is converted to a text string using the CONVERT phrase rules. Leading spaces in the resulting string are suppressed in the message. When *text* is numeric-edited, leading spaces are suppressed in the message, and the rest of the item is displayed without modification. For all other data types, *text* is displayed without modification.

4. *Title* is displayed in the message box’s title bar. If *title* is omitted, the message box displays the same title as the application’s main window.

5. *Type*, *icon*, *default* and *value* use a set of constants to describe the type of message box and the buttons it contains. These constants have level 78 definitions for them in the COPY library “acugui.def” supplied with the compiler. These constants are as follows:

```
78  MB-OK               VALUE 1.
78  MB-YES-NO           VALUE 2.
78  MB-OK-CANCEL        VALUE 3.
78  MB-YES-NO-CANCEL    VALUE 4.
78  MB-YES              VALUE 1.
78  MB-NO               VALUE 2.
78  MB-CANCEL           VALUE 3.
78  MB-DEFAULT-ICON     VALUE 1.
78  MB-WARNING-ICON     VALUE 2.
78  MB-ERROR-ICON       VALUE 3.
```

6. *Type* describes the set of buttons contained in the box. The possible values are:

Type Value	**Buttons**
MB-OK	“OK” button
MB-YES-NO	“Yes” and “No” buttons

Type Value	Buttons
MB-OK-CANCEL	"OK" and "Cancel" buttons
MB-YES-NO-CANCEL	"Yes", "No" and "Cancel" buttons

If *type* is omitted, or if it contains an invalid value, then MB-OK is used. The text that appears on the buttons is set by the current language installed for Windows. For non-Windows systems, the text is configurable by the TEXT configuration entry.

7. *Icon* describes the icon that will appear. The icon appears only under Windows. On other systems, the icon selected is ignored. If *icon* is set to MB-ERROR-ICON, then a "stop" icon is shown. If *icon* is set to MB-WARNING-ICON, then an "exclamation" icon displays. If *icon* is set to MB-DEFAULT-ICON, then boxes with "OK" buttons will display an "information" icon, and boxes with "Yes/No" buttons will display a "question mark" icon. If *icon* is omitted or contains an invalid value, then MB-DEFAULT-ICON is used.

8. *Default* describes which button will be the default button (i.e., the button used if the user simply presses "return"). The possible values are:

Value	Button
MB-OK	"OK" button
MB-YES	"Yes" button
MB-NO	"No" button
MB-CANCEL	"Cancel" button

If *default* is omitted, or contains an invalid value, then the default button will be the "OK" button or the "Yes" button.

9. *Value* will contain the identity of the button the user pressed to leave the message box. It uses the same values as *default* does, described above. For example, if the user presses the "No" button, then *value* will contain MB-NO.

Note: For each invocation of the message box, only one keystroke entry is picked up. If you want to drive the message box from your defined keystroke file (or from the COPY library "acugui.def"), make sure that you make this one entry a valid character for the message box, such as "Y" or "N" for a message requiring a "Yes" or a "No" response.

DISPLAY external-form-item

Format 16

DISPLAY external-form-item merges data into an HTML template file and sends the result to standard output.

```
DISPLAY external-form-item
```

Syntax Rules

external-form-item is an output record for an HTML form when used in a Common Gateway Interface (CGI) program. It is a group data item, declared with the IS EXTERNAL-FORM and IDENTIFIED BY clauses. It may have one or more elementary items associated with HTML template fields. The association is made with the IS IDENTIFIED BY clause.

external-form-item may also be an input record for an HTML form. In this case, the group item is declared with only the IS EXTERNAL-FORM clause. This is used primarily when you are debugging your CGI program. See the General Rules below for more details. See **section 5.7.1, "Data Description Entry,"** for information about how to declare external forms.

General Rules

1. An "external form" is called an "output form" if the IDENTIFIED BY clause is used to associate it with an HTML template file. If the IDENTIFIED BY clause is omitted, it is called an "input form". For example, the following is an input form:

```
01  CGI-FORM IS EXTERNAL-FORM.
    03  CGI-VAR1  PIC X(10).
    03  CGI-VAR2  PIC X(10).
```

And here is an output form:

```
01  HTML-FORM IS EXTERNAL-FORM
      IDENTIFIED BY "tmplate1".
    03  HTML-VAR1 PIC X(10).
    03  HTML-VAR2 PIC X(10).
```

The DISPLAY verb treats input and output forms differently. Input forms are discussed in rule 7, below. For output forms, DISPLAY merges the data contained in the elementary items into the associated HTML template file and sends the result to the standard output stream in conformance with the CGI specification. To do this, DISPLAY scans the HTML template file for data names delineated by two percentage signs on either side (i.e. %%data-name%%). It then replaces those data names with the contents of the associated elementary items from the output form, stripping trailing spaces.

2. The maximum length of a single line in the template file is 256 bytes. The maximum length of a single HTML output line is 512 bytes. No conversion is performed on the output form items before they are merged with the HTML template file.

3. You may specify a series of directories for locating HTML template files. To do this, use the **HTML_TEMPLATE_PREFIX** configuration variable. This variable is similar to FILE-PREFIX and CODE-PREFIX. It specifies a series of one or more directories to be searched for the desired HTML template file. The directories are specified as a sequence of space-delimited prefixes to be applied to the file name. All directories in the sequence must be valid names. The current directory can be indicated by a period (regardless of the host operating system).

   You may omit the template file suffix if it is either ".html" or ".htm". If the suffix is omitted or is something other than ".html" or ".htm", DISPLAY first appends ".html" to the specified file name and tries to open it. If that fails, DISPLAY appends ".htm" to the file name and tries to open it. If that fails, DISPLAY tries to open the file exactly as specified. If these attempts fail, the following error message is sent to the standard output stream in HTML format:

   ```
   Can't open HTML template "template-file-name"
   ```

When the Web Server executes your CGI program, the current working directory depends on the configuration of the specific Web Server that is running. In many cases the current working directory is the same as the Web Server's "root" directory. As part of the CGI specification, when the Web Server executes your CGI program, it sets an environment variable called PATH_TRANSLATED to the directory containing your CGI program. You may want to use this information to locate your HTML template files. For example, if your template files are in the same directory as your CGI programs, then set the HTML-TEMPLATE-PREFIX configuration variable to the value of PATH_TRANSLATED as follows:

```
01  CGI-DIRECTORY  PIC X(100) VALUE SPACES.

...

ACCEPT CGI-DIRECTORY FROM ENVIRONMENT "PATH_TRANSLATED".

SET CONFIGURATION "HTML-TEMPLATE-PREFIX" TO CGI-DIRECTORY.
```

The output from a CGI program must begin with a "response header". DISPLAY automatically generates a "Content-Type" response header if the specified template file is a local file (i.e., not a URL--see rule 5 below).

4. You may specify the EXTERNAL-FORM clause for an item that has no subordinate items. This is useful for displaying static Web pages. To do this, specify the name of the static Web page in the IDENTIFIED BY clause. For example, if you have a Web page called "webpage1.html", add the following lines to your COBOL program:

```
01  WEB-PAGE-1 IS EXTERNAL-FORM
       IDENTIFIED BY "webpage1".

...

    DISPLAY WEB-PAGE-1.
```

5. You may also specify a complete URL instead of a template file name in the IDENTIFIED BY clause. In this case, DISPLAY generates a "Location" response header that contains the URL. This header specifies that the data you're returning is a pointer to another location.

To determine whether the template file name is a URL, DISPLAY scans it for the "://" string. DISPLAY does not apply the HTML-TEMPLATE-PREFIX when the template file name is a URL.

For example, if your program determines that the information the user has requested is on another Web server, and its URL is "http://www.theinfo.com", add the following lines to your COBOL program:

```
01 THE-INFO-URL IS EXTERNAL-FORM
       IDENTIFIED BY "http://www.theinfo.com"

...

    DISPLAY THE-INFO-URL.
```

The length of the URL must not exceed 256 bytes.

Only one response header is sent to the standard output stream. Your CGI program should exit immediately after sending a location header (i.e., after displaying an external form identified by a URL).

6. You may use as many HTML template files as you like in a single program. A common way to use multiple HTML template files is to have three output forms: a header, body, and footer. Each of these has a corresponding HTML template file. You first display the header form, then move each row of data to the body form and display it, and finally display the footer form.

7. When an input form is specified in a DISPLAY statement, the names and values of each elementary item are sent to the standard output stream in HTML format. One line is generated for each elementary item. The line consists of the name of the item followed by " = ", followed by the first 100 bytes of the item's value. This can be useful when you are testing and debugging your CGI program.

DISPLAY UPON ENVIRONMENT-NAME

Format 17

DISPLAY UPON ENVIRONMENT-NAME sets the value of the specified environment variable in the ENVIRONMENT-NAME register.

```
DISPLAY name UPON ENVIRONMENT-NAME
```

Syntax Rules

name is an alphanumeric Working-Storage data item that is the name of an environment variable or configuration variable.

General Rules

DISPLAY UPON ENVIRONMENT-NAME sets ENVIRONMENT-NAME to the value of the environment variable or configuration variable specified by *name*. The value of ENVIRONMENT-NAME can be queried with a Format 13 ACCEPT statement.

DISPLAY assembly-name

Format 18

DISPLAY assembly-name instantiates a graphical .NET control or assembly.

```
DISPLAY {assembly-name       }
        {OBJECT assembly-name}

NAMESPACE { IS } "namespace"

CLASS-NAME { IS } "class-name"
```

Remaining phrases are optional.

```
HANDLE { IS } handle-1

VERSION { IS } "version"

CULTURE { IS } "culture"

STRONG-NAME { IS } "strong-name"

CONSTRUCTOR { IS } CONSTRUCTOR[n] parameters...

MODULE { IS } "module"
```

```
FILE-PATH { IS } "file-path"

AT LINE NUMBER line-num   [CELL  ]
                          [CELLS ]
                          [PIXEL ]
                          [PIXELS]

AT {COLUMN  } NUMBER col-num   [CELL  ]
   {COL     }                  [CELLS ]
   {POSITION}                  [PIXEL ]
   {POS     }                  [PIXELS]

SIZE  {IS} length [CELL  ]
      {= }        [CELLS ]
                  [PIXEL ]
                  [PIXELS]

LINES {IS} height [CELL  ]
      {= }        [CELLS ]
                  [PIXEL ]
                  [PIXELS]
```

Syntax Rules

1. Assembly-name is the name of a .NET assembly defined in a COPY file created by NETDEFGEN. This must be the DLL name of a graphical control, not an executable file. Graphical controls are generated by Visual Studio when a developer selects a “Windows Control Library” project type.

2. *Handle-1* is a USAGE HANDLE or PIC X(10) data item assigned to the graphical .NET assembly. The HANDLE phrase can be specified anywhere in the statement after the initial elements.

3. A value surrounded by quotation marks is an alphanumeric literal and is case-sensitive. Literal values for assembly parameters are located in the COPY file generated by NETDEFGEN. The same COPY file must be included in the SPECIAL-NAMES paragraph of your program.

4. *Line-num*, *col-num*, *length*, and *height* are numeric data items or literals. They may be non-integer values, except when pixels are specified. They may be numeric expressions.

General Rules

1. The Format 18 DISPLAY statement is only for graphical .NET assemblies. Graphical assemblies show the keyword "VISUAL" in the COPY file after the CLASS keyword. If the word "VISUAL" does not appear, use the CREATE statement to instantiate the assembly.
2. *Assembly-name* is the name of a .NET assembly defined in the NETDEFGEN COPY file. This must be a DLL name or a control name, not an executable.
3. *Namespace* is a NameSpace in the assembly, as it appears in the COPY file.
4. *Class-name* is a class in the NameSpace.
5. *Version* is the version number of the assembly.
6. *Culture* is cultural information related to the assembly.
7. *Strong-name* is the cryptographic key required to access the assembly, if any. If the assembly requires such a key, as all assemblies in the Global Assembly Cache (GAC) do, it is shown in the COPY file under the keyword STRONG.
8. All classes that result in an object have a CONSTRUCTOR, which is a sort of method. If you see a CONSTRUCTOR identifier in the COPY file without a parameter list, then the field may be omitted from your COBOL program. If all listed CONSTRUCTORs have parameters, then you must choose which CONSTRUCTOR and parameters to use. *Constructor*(n) is the constructor that you want to use followed by its parameter data.
9. *Module* identifies a file where a combination of NameSpaces and Classes reside. It is used when the assembly is constructed of other assemblies.
10. *File-path* is the path of an XML file, and that XML file contains the path where the .NET assembly is located. Use FILE-PATH when the assembly that you want to access does not reside in the GAC or in the same directory as "wrun32.exe". Assemblies that reside in the GAC will have the STRONG keyword in the NETDEFGEN COPY file.
11. LINES and SIZE default to the design control height and width.

DIVIDE Statement

The DIVIDE statement performs arithmetic division.

General Format

Format 1

```
DIVIDE divisor INTO { result [ROUNDED] } ...

   [ ON SIZE ERROR statement ]

   [ NOT ON SIZE ERROR statement ]

   [ END-DIVIDE ]
```

Format 2

```
DIVIDE divisor INTO dividend

     GIVING { result [ROUNDED] } ...

   [ ON SIZE ERROR statement ]

   [ NOT ON SIZE ERROR statement ]

   [ END-DIVIDE ]
```

Format 3

```
DIVIDE dividend BY divisor

     GIVING { result [ROUNDED] } ...

   [ ON SIZE ERROR statement ]

   [ NOT ON SIZE ERROR statement ]

   [ END-DIVIDE ]
```

Format 4

```
DIVIDE divisor INTO dividend GIVING result [ROUNDED]

     REMAINDER remainder

   [ ON SIZE ERROR statement ]
```

```
   [ NOT ON SIZE ERROR statement ]

   [ END-DIVIDE ]
```

Format 5

```
DIVIDE dividend BY divisor GIVING result [ROUNDED]

     REMAINDER remainder

   [ ON SIZE ERROR statement ]

   [ NOT ON SIZE ERROR statement ]

   [ END-DIVIDE ]
```

Syntax Rules

1. *Divisor* is a numeric literal or numeric data item. It represents the number to be divided by.
2. *Dividend* is a numeric literal or numeric data item. It represents the number to be divided into.
3. *Result* is a numeric or numeric-edited data item. In Format 1, *result* must be a numeric data item.
4. *Remainder* is a numeric or numeric-edited data item.
5. *Statement* is an imperative statement.
6. The REMAINDER phrase may not be used if any operand or *result* is an external floating-point data type.

Note: Avoid mixing external floating-point types with other numeric types in DIVIDE statements, because mixed-type DIVIDE operations are not completely reliable.

General Rules

Format 1

The *divisor* is individually divided into each *result* and the quotient stored there.

Format 2 and 3

The *divisor* is divided into the *dividend*. The quotient is stored in each *result*.

Format 4 and 5

1. The *divisor* is divided into the *dividend*, the quotient is stored in *result*, and the remainder is stored in *remainder*.
2. If the ROUNDED phrase is not specified, the quotient is truncated to fit into the *result* and the *remainder* is computed by subtracting the product of the truncated quotient and *divisor* from the *dividend*. If the ROUNDED phrase is specified, the quotient is rounded to fit the *result*, but the *remainder* is still computed by subtracting the product of the truncated quotient and the *divisor* from the *dividend*.
3. If the SIZE ERROR clause is specified, and a size error occurs during the computation of the quotient, the remainder is not computed and *remainder* remains unchanged. If a size error occurs during the computation of the remainder, the *result* is updated, but the remainder is left unchanged. The SIZE ERROR statement executes in either case.

Note: In division operations, the remainder is calculated before the quotient is moved to the destination item(s).

The remainder will almost always be 0 if the dividend or divisor is floating-point. This is because all of the arithmetic is performed using floating-point variables. The remainder will be non-zero only if precision is lost during the calculation.

---

4. Additional information can be found in the sections covering **Arithmetic Operations** (6.4.1), **Multiple Receiving Fields** (6.4.2), the **ROUNDED Option** (6.4.3), and the **SIZE ERROR Option** (6.4.4).

ENTRY Statement

The ENTRY statement is used to establish an alternate entry point in the program.

General Format

```
ENTRY entry-name [ USING {parameter} ... ]
```

Syntax Rules

1. *entry-name* is a non-numeric literal. The maximum number of characters is 30.
2. *parameter* is a data item defined with a level-number of 01 or 77 and it must appear in the Linkage Section.
3. Each *parameter* cannot appear more than once in the USING phrase.
4. An ENTRY statement can begin in Area A. The compiler will not recognize this as an error. This provides compatibility with other compilers that support the ENTRY statement.

General Rules

1. *entry-name* must be unique within the program. No duplicates are allowed.
2. The maximum number of parameters that may be passed to an ENTRY point is 255.
3. If the USING phrase is used, then each *parameter* must be declared in the Linkage Section.
4. The constraints on the parameters are the same as those for the Procedure Division USING items. See **Section 6.5, "Procedure Division Format,"** for details.
5. There is a limit of 255 level 01 Linkage data items per program.
6. An ENTRY point can be called just like any other program; however, the program containing the ENTRY point must be loaded and present in memory at the time of the CALL. Once loaded, COBOL objects

may be called by any of their ENTRY point names. See *ACUCOBOL-GT User's Guide*, section 2.9, "Calling Subprograms," for details.

7. The CALL succeeds if the specified *program-name* finds an exact *entry-name* match. By default, the matching logic is case sensitive and distinguishes between hyphens and underscores. The matching logic can be configured to be case insensitive and to not distinguish between hyphens and underscores with the **LITERAL_ENTRY** configuration variable described in Appendix H.

8. When a program is called by one of its ENTRY point names, the execution begins immediately after the ENTRY statement. The ENTRY point parameters are initialized in the same way as the Procedure Division parameters.

Note: Care should be taken not to enter a COBOL program in the middle of a control flow statement like a loop or a conditional statement. Doing so will result in undefined behavior.

---

9. Execution of the program passes through entry points with no effect.

10. There is a limit of 65536 ENTRY points per program, including the main entry point at the beginning of the Procedure Division.

EVALUATE Statement

The EVALUATE statement causes multiple conditions to be evaluated. The subsequent action of the program depends on the results of these evaluations.

The EVALUATE statement is very similar to the CASE construct common in many other programming languages. The EVALUATE/CASE construct provides the ability to selectively execute one of a set of instruction alternatives based on the evaluation of a set of choice alternatives.

EVALUATE extends the power of the typical CASE construct by allowing multiple data items and conditions to be named in the EVALUATE phrase (see code example 2).

Note: This manual entry includes code examples and highlights for first-time users following the General Rules section.

General Format

```
EVALUATE  {subject}  [ ALSO {subject} ] ...
          {TRUE   }         {TRUE   }
          {FALSE  }         {FALSE  }

  { { WHEN obj-phrase [ ALSO obj-phrase ] ... } ...

           statement-1 } ...

  [ WHEN OTHER statement-2 ]

  [ END-EVALUATE ]
```

obj-phrase has the following format:

```
  { ANY                                  }
  { TRUE                                 }
  { FALSE                                }
  { [=] cond-obj                         }
  { [NOT =] obj-item [ {THRU   } obj-item ]}
  {                    {THROUGH}         }
  { < obj-item                           }
  { IS LESS THAN obj-item                }
  { <= obj-item                          }
  { IS LESS THAN OR EQUAL TO obj-item    }
  { = obj-item                              }
  { IS EQUAL TO obj-item                    }
  { EQUALS obj-item                         }
  { > obj-item                              }
  { IS GREATER THAN obj-item                }
  { EXCEEDS obj-item                        }
  { >= obj-item                             }
  { IS GREATER THAN OR EQUAL TO obj-item    }
  { <> obj-item                             }
  { IS UNEQUAL TO obj-item                  }
```

Syntax Rules

1. *Subject* may be a literal, data item, arithmetic expression, or conditional expression.
2. *Cond-obj* is a conditional expression.
3. *Obj-item* may be a literal, data item, or arithmetic expression.
4. *Statement-1* and *statement-2* are imperative statements.
5. Before the first WHEN phrase, *subject* and the words TRUE and FALSE are called "subjects," and all the subjects together are called the "subject set".
6. The operands and the words TRUE, FALSE, and ANY which appear in a WHEN phrase are called "objects," and the collection of objects in a single WHEN phrase is called the "object set".
7. The words THROUGH and THRU are equivalent. Two *obj-items* connected by a THROUGH phrase must be of the same class. They are treated as a single object.
8. The number of objects within each object set must match the number of subjects in the subject set.
9. Each object within an object set must correspond to the subject having the same ordinal position as in the subject set. For each pair:
    a. *Obj-item* must be a valid operand for comparison to the corresponding *subject*.
    b. TRUE, FALSE, or *cond-obj* as an object must correspond to TRUE, FALSE, or a conditional expression as the subject.
    c. ANY may correspond to any type of subject.

General Rules

1. The EVALUATE statement operates as if each subject and object were evaluated and assigned a value or range of values. These values may be numeric, nonnumeric, truth values, or ranges of numeric or nonnumeric values. These values are determined as follows:

a. Any subject or object that is a data item or literal, without either the THROUGH or the NOT phrase, is assigned the value and class of that data item or literal.

b. Any subject or object that is an arithmetic expression, without either the THROUGH or the NOT phrase, is assigned a numeric value according to the rules for evaluating arithmetic expressions.

c. Any subject or object that is a conditional expression is assigned a truth value according to the rules for evaluating conditional expressions.

d. Any subject or object specified by the words TRUE or FALSE is assigned a truth value corresponding to that word.

e. Any object specified by the word ANY is not evaluated.

f. If the THROUGH phrase is specified for an object, without the NOT phrase, the range of values includes all permissible values of the corresponding subject that are greater than or equal to the first operand and less than or equal to the second operand, according to the rules for comparison.

g. If the NOT phrase is specified for an object, the values assigned to that object are all permissible values of the corresponding subject not equal to the value, or range of values, that would have been assigned had the NOT phrase been omitted.

2. The EVALUATE statement then proceeds as if the values assigned to the subjects and objects were compared to determine if any WHEN phrase satisfies the subject set. Each object within the object set for the first WHEN phrase is compared to the subject having the same ordinal position within the subject set. The comparison is satisfied if one of the following is true:

   a. If the items being compared are assigned numeric or nonnumeric values, the comparison is satisfied if the value (or one of the range of values) assigned to the object is equal to the value assigned to the subject.

   b. If the items being compared are assigned truth values, the comparison is satisfied if the truth values are the same.

   c. If the object is the word ANY, the comparison is always satisfied.

3. If the comparison is satisfied for every object within the object set, the corresponding WHEN phrase is selected.

4. If the comparison is not satisfied for one or more objects within the object set, the procedure repeats for the next WHEN phrase. This is repeated until a WHEN phrase is selected or all the object sets have been tested.

5. If a WHEN phrase is selected, the corresponding *statement-1* is executed.

6. If no WHEN phrase is selected and a WHEN OTHER phrase is specified, *statement-2* is executed. If no WHEN OTHER phrase is present, control transfers to the end of the EVALUATE statement.

7. The WHEN verb is accepted as an implied END-IF or END-PERFORM for any and all preceding IF and PERFORM statements that do not have corresponding END- statements.

8. The scope of execution of the EVALUATE statement is terminated when the end of *statement-1* or *statement-2* is reached, or when no WHEN phrase is selected and no WHEN OTHER phrase is specified.

Code Examples

Example 1:

```
EVALUATE AGE
   WHEN 56 THRU 99  PERFORM  SENIOR_PROSPECT
   WHEN 40 THRU 55  PERFORM  MATURE_PROSPECT
   WHEN 21 THRU 39  PERFORM  YOUNG_PROSPECT
   WHEN OTHER       PERFORM  NOT_A_PROSPECT
END-EVALUATE.
```

Example 2:

```
EVALUATE INCOME ALSO TRUE
   WHEN  20000 THRU 39999  ALSO RISK_CLASS = "A"
      PERFORM LOW_INCOME_PROSPECT
   WHEN  40000 THRU 59999  ALSO RISK_CLASS = "A"
      PERFORM  MID_INCOME_PROSPECT
   WHEN  60000 THRU 999999 ALSO RISK_CLASS = "A"
      PERFORM  HIGH_INCOME_PROSPECT
   WHEN  60000 THRU 999999 ALSO NOT RISK_CLASS = "A"
      PERFORM  HIGH_INCOME_HIGH_RISK_PROSPECT
```

```
   WHEN  OTHER
      PERFORM UNCLASSIFIED_PROSPECT
END-EVALUATE.
```

Highlights for first-time users

1. Statement subjects (associated with the EVALUATE phrase) and statement objects (associated with the WHEN phrase) must be equal in number, correspond by position and be valid operands for comparison. Note the number and order of subjects in example 2 and the correspondent number and position of WHEN objects.

2. If all of the conditions in a WHEN phrase match, the associated imperative statement is executed. None of the remaining WHEN phrases is evaluated. Program execution then falls through to the end of the EVALUATE statement.

3. The WHEN OTHER phrase is an optional phrase for the handling of all remaining cases (the set of possible conditions not explicitly tested for by the preceding WHEN phrases). The WHEN OTHER phrase, if present, must be the last WHEN phrase in the statement.

4. The words TRUE and FALSE may be used in the subject or object phrase to specify a literal truth condition.

5. The word ANY may be used in the WHEN phrase to specify an unconditional match with the corresponding item in the subject phrase.

6. The word NOT may be used in the WHEN phrase to negate its associated condition.

7. The word THROUGH or THRU may be used in the WHEN phrase to describe a range of values. When combined with NOT, THRU describes an excluded set of values. For example, `NOT 10 THRU 20` means that any object holding a value from 10 to 20, including the numbers 10 and 20, will result in a FALSE, or no match evaluation.

EXHIBIT Statement

The EXHIBIT statement causes an (optionally conditional) display of the literals, and/or variables (optionally preceded by the variable name) specified in the statement. This statement is only supported when a program is compiled in IBM OSVS compatibility mode (-Cv or -Cv=OSVS).

General Format

```
EXHIBIT [NAMED] [CHANGED] {literal | variable} ...
```

General Rules

1. If neither NAMED nor CHANGED is used, each literal and variable value is displayed.
2. If only NAMED is used, each literal is displayed, and each variable is displayed. Variables are preceded by "variable-name=" (where "variable-name" is replaced with the name of the variable in the EXHIBIT statement).
3. If only CHANGED is used, each literal is displayed, and each variable is displayed if its value is different from the last time this EXHIBIT verb was executed.
4. If both NAMED and CHANGED are used, each literal is displayed, and each variable is displayed if its value is different from the last time this EXHIBIT verb was executed. In addition, each variable is preceded by "variable-name=" (where "variable-name" is replaced with the name of the variable in the EXHIBIT statement).
5. As a compatibility issue, its recommended that you modify your source code to use actual DISPLAY statements, and that you not add new EXHIBIT statements to your COBOL program.

EXIT Statement

The EXIT statement returns control to a calling program or provides a common logical end point for a series of procedures.

Format 1

```
EXIT
```

Format 2

```
EXIT PROGRAM [ {RETURNING} return-value ]
               {GIVING   }
```

Format 3

```
EXIT PERFORM [ CYCLE ]
```

Format 4

```
EXIT  {PARAGRAPH}
      {SECTION  }
```

Syntax Rules

1. A Format 1 EXIT statement must be in a sentence by itself, and that sentence must be the only sentence in its paragraph.
2. *Return-value* must be a numeric literal or data item.
3. An EXIT PERFORM statement must occur within the scope of an in-line PERFORM statement.
4. An EXIT SECTION statement must be contained within a section.

General Rules

Format 1

A Format 1 EXIT statement associates a paragraph name with a point in the program. It has no effect on program execution. A paragraph containing a Format 1 EXIT statement is equivalent to an empty paragraph.

Format 2

1. An EXIT PROGRAM statement has no effect if executed in a program that is the first program of a thread or any program that was not called by another. Neither does it have any effect if it is executed within the scope of an EVENT procedure, unless the return point is also within the scope of the EVENT procedure.

2. In a called program, the EXIT PROGRAM statement causes the current program to exit, and execution resumes at the next executable statement after the CALL statement in the calling program.

3. If the exiting program has the initial attribute, it is immediately canceled (see the entry in this section for the "CANCEL Statement"). If it does not have the initial attribute, it retains its current state the next time it is called.

4. If *return-value* is specified, then it is assigned to the special register RETURN-CODE before the program is exited. This special register is defined as:

```
77  RETURN-CODE SIGNED-LONG, EXTERNAL.
```

   It is implicitly shared by all programs of a run unit and is automatically created by the compiler. The final value of RETURN-CODE is returned to the host operating system when the run unit completes.

   The compiler also creates an unsigned version of the return code called RETURN-UNSIGNED. It has the following implied definition:

```
77  RETURN-UNSIGNED
        REDEFINES RETURN-CODE UNSIGNED-LONG, EXTERNAL.
```

Format 3

1. An EXIT PERFORM statement causes control to pass to a point just past the END-PERFORM that matches the innermost PERFORM statement containing the EXIT PERFORM. This causes the program to jump out of the innermost in-line PERFORM.

2. If the CYCLE option is given, then control is passed to a point just *prior* to the matching END-PERFORM instead. For PERFORM constructs that imply looping, this will cause control to pass to the next iteration of the loop (note that the loop-terminating condition will be tested).

Format 4

1. An EXIT PARAGRAPH statement causes control to pass to an imaginary CONTINUE statement placed at the end of the current paragraph.

2. An EXIT SECTION statement causes control to pass to an imaginary CONTINUE statement placed at the end of the last paragraph in the current section.

GOBACK Statement

The GOBACK statement exits the current program regardless of whether or not it is a called program.

General Format

```
GOBACK { [ {RETURNING} return-value] }
         {GIVING   }
```

Syntax Rules

1. If a GOBACK statement is in a consecutive sequence of imperative statements in a sentence, it must be the last statement in that sentence.
2. *Return-value* must be a numeric literal or data item.

General Rules

1. The GOBACK statement is equivalent to the statement sequence

   ```
   EXIT PROGRAM; STOP RUN
   ```

   This causes the current program to return to the caller if it is a called program or causes the run unit to halt if the program is not a called program.

2. If *return-value* is specified, then it is assigned to the special register RETURN-CODE before the program is exited. This special register is defined as:

   ```
   77  RETURN-CODE  SIGNED-LONG, EXTERNAL.
   ```

   It is implicitly shared by all programs of a run unit and is automatically created by the compiler. The final value of RETURN-CODE is returned to the host operating system when the run unit completes.

The compiler also creates an unsigned version of the return code called RETURN-UNSIGNED. It has the following implied definition:

```
77 RETURN-UNSIGNED
      REDEFINES RETURN-CODE UNSIGNED-LONG, EXTERNAL.
```

GO TO Statement

The GO TO statement provides for a direct transfer of control in the Procedure Division.

General Format

Format 1

```
GO TO procedure-name
```

Format 2

```
GO TO {procedure-name} ... DEPENDING ON depend-item
```

Syntax Rules

1. *Procedure-name* is the name of a paragraph or section in the program.
2. *Depend-item* is an integer elementary numeric data item.
3. A Format 1 GO TO that is in a consecutive sequence of imperative statements in a sentence must be the last statement in that sentence.

General Rules

Format 1

The GO TO statement transfers control to *procedure-name*. No return mechanism is implied.

Format 2

1. The GO TO DEPENDING statement transfers control to one of the *procedure-names* depending on the value of *depend-item*. A *depend-item* value of "1" refers to the first *procedure-name*, a value of "2" refers to the second, and so on.

2. If *depend-item* is less than or equal to zero, or is greater than the number of *procedure-names*, no control transfer occurs. In this case, the GO TO statement has no effect.

IF Statement

The IF statement provides for conditional action by the program.

General Format

```
IF condition THEN { {statement-1} }
                  { NEXT SENTENCE }

    [ ELSE {statement-2 } [END-IF] ]
    [ ELSE NEXT SENTENCE           ]
    [ END-IF                       ]
```

Syntax Rules

1. *Statement-1* and *statement-2* are imperative or conditional statements. An imperative statement can precede a conditional statement.

2. *Condition* is any conditional expression.

3. The ELSE NEXT SENTENCE phrase is optional if it immediately precedes a period ending a sentence.

4. If END-IF is specified, NEXT SENTENCE must not be specified.

General Rules

1. The IF statement provides a method for selecting alternate sets of statements to execute depending on the truth value of *condition*.

2. The scope of an IF statement ends when one of the following is encountered:

   a. A period ending a sentence.

   b. An END-IF phrase at the same nesting level.

   c. An ELSE phrase associated with an IF statement at a higher nesting level.

3. If *condition* is "true", then *statement-1* executes. *Statement-2* is not executed. If NEXT SENTENCE is used instead of *statement-1*, control immediately passes to the next executable sentence. Note that the ANSI standard states that "NEXT SENTENCE is an archaic feature and its use should be avoided."

4. If *condition* is "false", *statement-2* executes. *Statement-1* is not executed. If the ELSE NEXT SENTENCE phrase is used, control immediately passes to the next executable sentence. If the ELSE phrase is not present, then control passes to the end of the IF statement.

5. OTHERWISE is a synonym for ELSE in the IBM DOS/VS COBOL "-Cv" compatibility mode. See Chapter 5, "IBM DOS/VS COBOL Conversions," in *Transitioning to ACUCOBOL-GT* for more information.

INITIALIZE Statement

The INITIALIZE statement sets selected types of elementary data items to chosen values.

General Format

```
INITIALIZE { destination } ... [ WITH FILLER ]

[ REPLACING { {ALPHABETIC         } DATA BY value }...]
            {ALPHANUMERIC       }
            {NUMERIC            }
            {ALPHANUMERIC-EDITED}
            {NUMERIC-EDITED     }
```

Syntax Rules

1. *Destination* is a data item.
2. *Value* is a literal or a data item. It must be a legal source for a MOVE to the corresponding category of data. For example, a *value* appearing in a REPLACING ALPHABETIC BY clause must have a category of alphabetic, alphanumeric, or alphanumeric-edited.
3. The same category cannot be repeated in a REPLACING phrase.
4. *Destination* may not be an index data item.
5. *Destination* may not contain a RENAMES clause.

General Rules

1. Whether *destination* references an elementary item or a group item, all operations are performed as if a series of MOVE statements had been written, each of which has an elementary item as its receiving field according to the following rules:
   a. If *destination* is a group item, any elementary item contained in *destination* is initialized only if it belongs to a category specified by the REPLACING phrase.
   b. If *destination* is an elementary item, that item is initialized only if it belongs to a category specified in the REPLACING phrase.
   c. Each data item that is initialized is treated as the receiving operand of an implicit MOVE statement with the corresponding *value* as the sending field.
   d. All elementary receiving fields, including all table occurrences, are affected, except as specified in the following rules.
2. Index data items and elementary FILLER data items are not affected by the INITIALIZE statement, unless the optional WITH FILLER phrase is specified, in which case FILLER data items are initialized.
3. Any item that is subordinate to *destination* and which contains a REDEFINES clause, or any item that is subordinate to such an item, is excluded from this operation. *Destination* itself, however, may be subordinate to a REDEFINES clause.

4. If no REPLACING phrase is specified, alphabetic, alphanumeric, and alphanumeric-edited data items are initialized to SPACES; numeric and numeric-edited data items are initialized to ZEROS.
5. If multiple *destinations* are specified, they are initialized in the order written.
6. *Destinations* that are or contain OCCURS DEPENDING ON items will use the maximum number of occurrences when being initialized.

INQUIRE Statement

The INQUIRE verb allows you to retrieve information from a control, or retrieve the dimensions of a window. General Format

Format 1

```
INQUIRE { control-item } [ ( {index-1} ... ) ]
        { CONTROL      }
```

Remaining phrases are optional, can appear in any order.

```
AT screen-loc   [CELL  ]
                [CELLS ]
                [PIXEL ]
                [PIXELS]

AT LINE NUMBER line-num   [CELL  ]
                          [CELLS ]
                          [PIXEL ]
                          [PIXELS]

AT {COLUMN  } NUMBER col-num   [CELL  ]
   {COL     }                  [CELLS ]
   {POSITION}                  [PIXEL ]
   {POS     }                  [PIXELS]

AT CLINE NUMBER cline-num   [CELL ]
                            [CELLS]

AT CCOL NUMBER ccol-num   [CELL ]
                          [CELLS]
```

```
TITLE {IN} title
      {= }

VALUE {IN} [ MULTIPLE ] value [ LENGTH {IN} length-1 ]
      {= } [ TABLE    ]                {= }

STYLE {IN} style-flags
      {= }

HELP-ID {IN} help-id
        {= }

{{property-name         } [({param-expr}...)] {IN} property-value ...
 {PROPERTY property-name}                     {= }
 {object-expression     }
    [ LENGTH {IN} length-1 ] }
             {= }

SYSTEM HANDLE {IN} system-handle
              {= }

POP-UP MENU {IN}  {menu-1}
            {= }

LINE NUMBER {IN} line-num
            {= }

{COLUMN  } NUMBER {IN} col-num
{COL     }        {= }
{POSITION}
{POS     }

SIZE {IN} width
     {= }

LINES {IN} height
      {= }

MAX-HEIGHT {IN} max-height
           {= }

MAX-WIDTH  {IN} max-width
           {= }
```

```
MIN-HEIGHT {IN} min-height
           {= }

MIN-WIDTH  {IN} min-width
           {= }

ID {IN} id
   {= }

CLASS {IN} class-code
      {= }

EXCLUDE-EVENT-LIST {IN} list-state
                   {= }

LAYOUT-DATA {IN} layout-data
            {= }

ENABLED {IN} enabled-state
        {= }

VISIBLE {IN} visible-state
        {= }
```

where *param-expr* is one of the following:

```
{ param                                 } [ AS type_num ]

{ {BY} NAME parameter-name {IS} param }
{                          {= }       }

{ parameter-name {IS} param           }
{                {= }                 }
```

object-expression has the following format:

```
{ {^} property-1 [ (param-expr ... ) ]
  [ :: property-2 [ ( param-expr ... ) ] ... }
```

Format 2

```
INQUIRE { window-handle           }
        { WINDOW [generic-handle] }
```

Remaining phrases are optional, can appear in any order.

```
LINE NUMBER {IN} line-no
            {= }

{COLUMN  } NUMBER {IN} col-num
{COL     }        {= }
{POSITION}
{POS     }

TITLE {IN} title
      {= }

SCREEN LINE NUMBER {IN} screen-line
                   {= }

SCREEN {COLUMN  } NUMBER {IN} screen-col
       {COL     }        {= }
       {POSITION}
       {POS     }

SIZE {IN} width
     {= }

LINES {IN} height
      {= }

SYSTEM HANDLE {IN} system-handle
              {= }

LAYOUT-MANAGER {IN} layout-manager
               {= }

VISIBLE {IN} visible-state
        {= }

POP-UP MENU {IN} menu-1
            {= }
```

Syntax Rules

1. *Control-item* is a USAGE HANDLE data item that identifies the control to be inquired. If it is a typed handle, then it must be associated with a control. *Control-item* can also be an elementary Screen Section item that describes a control.

2. *Index-1* is a numeric expression. The parentheses surrounding *index-1* are required.

3. The AT, LINE, COLUMN, CLINE, and CCOL phrases must appear in conjunction with the CONTROL phrase.

4. *Screen-loc* is an integer data item or literal that contains exactly 4, 6, or 8 digits.

5. *Line-num*, *col-num*, *cline-num*, and *ccol-num* are numeric data items or literals. Note that they may contain non-integer values, except when pixels are specified.

6. *Title* is an alphanumeric data item.

7. *Value* may be any data item.

8. *Style-flags* is a numeric data item capable of holding 10 or more digits.

9. *Help-id* is a numeric data item.

10. *Property-name* is the name of a property specific to the type of control being inquired. If *control-item* refers to a generic handle, or if the CONTROL option is specified, then *property-name* cannot be used. Use the PROPERTY phrase instead.

11. *Property-type* is a numeric literal or data item.

12. *Property-value* is a data item. Its data type should be appropriate for the specified property.

13. In *param-expr*:

    a. *Param* is a literal, data-item, or numeric expression used when inquiring the property value of an ActiveX control or COM object.

    b. *Type-num* is a numeric data item or numeric literal.

14. In *object-expression*:

    a. ^ can only be used in conjunction with a Format 5 USE verb for an ActiveX control or COM object.

    b. *Property-1* is the name of a property of the ActiveX control or COM object. *Property-1* cannot be a write-only property.

c. *Property-2* is the name of a property of the ActiveX control or COM object that is the value of *property-1*. *Property-2* cannot be a write-only property.

15. *Length-1* is a numeric data item. The LENGTH phrase may be specified only if the *value* or *property-value* immediately preceding it is an alphanumeric data item.

16. *Window-handle* is a USAGE HANDLE OF WINDOW or PIC X(10) data item.

17. *Generic-handle* is a USAGE HANDLE, HANDLE OF WINDOW, or PIC X(10) data item.

18. *Line-no*, *col-no*, *screen-line*, *screen-col*, *width*, *height*, *max-height*, *max-width*, *min-height*, and *min-width* are numeric data items. *Line-no* and *col-no* should be signed and have at least two digits after the decimal point to get the best results. *Screen-line* and *screen-col* should be signed to get the best result.

19. *System-handle*, *visible-state*, and *layout-data* are numeric data items.

20. *Menu-1* is a USAGE HANDLE or HANDLE OF MENU data item.

21. *Id*, *class-code* and *enabled-state* are numeric data items.

22. *List-state* is a numeric data item.

23. Layout-*manager* is a HANDLE or HANDLE OF LAYOUT-MANAGER data item.

24. In Format 1, the LINE IN phrase and the COLUMN IN phrase may be used only if the *control-item* option is specified.

General Rules

Format 1 (INQUIRE CONTROL)

1. The INQUIRE CONTROL statement retrieves some or all of a control's current properties, and stores them as data items. It can also be used to retrieve property data from ActiveX controls, COM objects, and .NET controls (also known as assemblies). *Control-item* identifies the control to inquire. If the CONTROL phrase is used instead, the runtime inquires the control located at the screen position specified by the AT, LINE, and

COLUMN phrases in the current window (on non-graphical systems, the CLINE and CCOL phrases also apply). The runtime system maintains a list of controls in each window. When you are attempting to inquire from a control at a specific location, the runtime searches this list, inquiring the first control it finds that exactly matches the given location. The list is maintained in the order in which the controls are created.

2. If *control-item* does not refer to a valid control, or if the runtime cannot locate a control at the specified screen location, the INQUIRE statement has no effect.

3. If *index-1* is specified, then certain properties in the control being inquired are modified to match the value of *index-1*. This modification occurs before any inquiry occurs. The exact set of properties modified depends on the control's type. Three controls have properties that are modified in this way:

Control Type	Properties Affected
List Box	QUERY-INDEX
Grid	Y, X
Tree View	ITEM

Each occurrence of *index-1* modifies one property. The first occurrence modifies the first property in the list presented in the preceding table. The second occurrence modifies the second property. For example, the statement fragment

```
INQUIRE grid-1(2, 3)
```

Would have the effect of setting the grid property "Y" to "2" and "X" to "3".

Supplying more index values than the control supports has no additional effect. You may omit trailing indexes; this leaves the corresponding properties unchanged.

This feature can be used to simplify inquiry on specific elements of controls that hold multiple values. For example, you can retrieve the contents of row 2, column 3 in a grid with the statement:

```
INQUIRE grid-1(2, 3), CELL-DATA IN data-1
```

This is exactly equivalent to the more cumbersome:

```
MODIFY grid-1, Y = 2, X = 3
INQUIRE grid-1, CELL-DATA IN data-1
```

4. When the runtime is storing data items, the rules for the MOVE statement are applied. The source for the title is alphanumeric. The type of control determines the source format of the value. The source format for a property is either numeric or alphanumeric depending on the specific property.

5. When used with an ActiveX control or COM object, INQUIRE gets the value of a property or gets the style flags.

6. When the PROPERTY phrase is used to set an ActiveX control or COM object, the runtime automatically converts parameters to the appropriate styles.

7. When the LENGTH option is specified, *length-1* gives you the exact number of characters that were placed by the control in *value* or *property-value*. This option is useful in determining how long the logical data is in *value* or *property-value*, or if there are trailing spaces. If, for example, you inquired the SELECTION-TEXT property in an entry-field and specified the LENGTH option, you could tell if the user's selection contains trailing spaces. If you do not use the LENGTH option, your program will not distinguish between the trailing spaces in the selection and the trailing spaces added by the runtime.

8. The SYSTEM HANDLE phrase retrieves the host graphical system's handle that corresponds to the control and stores this value in *system-handle*. This value is the way the host graphical system identifies the control. You usually need it if you want to affect the control from some other language such as C. There is no use for the host system's handle if you are using only ACUCOBOL-GT; the handle is useful only when you need to have another language interact with an ACUCOBOL-GT screen.

9. Each host system defines its own technique for identifying graphical components. Under Windows, the Windows API uses the "HWND" type, which is a 32-bit unsigned value. You can use UNSIGNED-INT as an appropriate USAGE type for *system-handle* to cover these two cases portably.

10. If the control does not have a corresponding host handle, then *system-handle* is set to zero. This indicates that either the host system does not have an underlying graphical system, or that the particular control does not use the host's notion of a control in its implementation.

11. The POP-UP MENU option returns the handle of the pop-up menu associated with the control in *menu-1*. If the control has no pop-up menu, *menu-1* is set to NULL.

12. The LINE NUMBER IN and COLUMN NUMBER IN phrases return the location of the control in *line-num* and *col-num* respectively. The SIZE IN and LINES IN phrases return the dimensions of the control in *width* and *height* respectively. These values have the same meaning and units that they have in a DISPLAY or MODIFY statement.

    Note: In order for the LINES IN and SIZE IN phrases to return a value, the control must have been *created* with its LINE and SIZE dimensions specified. Then the value returned is in the units used to create the control. If the control was not given dimensions when it was created, the INQUIRE statement has no effect.

    ---

13. The MAX-HEIGHT, MAX-WIDTH, MIN-HEIGHT, and MIN-WIDTH phrases return the value of the control's current maximum and minimum size restrictions in *max-height*, *max-width*, *min-height*, and *min-width*, respectively. These values have the same meaning and units that they have in a DISPLAY or MODIFY statement.

14. The ID IN phrase returns the control's ID if it has one. If it does not (or the control does not exist), *id* is set to zero.

15. The CLASS IN phrase returns the type of the control (label, entry field, etc). This is coded as a unique number for each class. The appropriate values can be found in the COPY library "controls.def".

16. The LAYOUT-DATA IN phrase sets *layout-data* to the current value of the control's LAYOUT-DATA property.

17. The ENABLED IN phrase sets *enabled-state* to "1" if the control is enabled and "0" if the control is disabled.

18. The VISIBLE IN phrase sets *visible-state* to "1" if the control is visible and "0" if it is invisible.

19. You cannot use named parameters to avoid entering required parameters. You can omit optional parameters only.

20. You must specify only unnamed parameters before the BY NAME clause, and only named parameters after the BY NAME clause.

21. You can use one- and two-dimensional COBOL tables as property and method parameters for use in COM SAFEARRAYs. The runtime automatically converts the table to an COM SAFEARRAY, as long as it contains only one elementary item that is USAGE HANDLE or USAGE HANDLE OF VARIANT. See section 4.3 in *A Guide to Interoperating with ACUCOBOL-GT.*

22. Use the "AS *type-num"* phrase in the parameter expression if you want to force the parameter to be converted to a particular VARIANT type before it is passed to a property or method of an ActiveX control or COM object. You can tell from the object's documentation and the name of the parameter whether the object expects a particular VARIANT type, such as boolean.

    Use the AS phrase if the ActiveX or COM object requires a method or property parameter to be something different from the default VARIANT type chosen by the runtime for the particular COBOL data item or literal. (See section 4.3 in *A Guide to Interoperating with ACUCOBOL-GT* for the rules that the runtime uses to determine the VARIANT type). Specify the word "AS" followed by a numeric literal or level 78 numeric constant that indicates the variant type to which you want the parameter converted. The "activex.def" COPY file in the ACUCOBOL-GT sample/def directory contains predefined level 78 constants for each of the VARIANT types.

Format 2 (INQUIRE WINDOW)

1. The INQUIRE WINDOW statement returns one or more attributes of the window identified by *window-handle* or *generic-handle*. If the WINDOW phrase is used and *generic-handle* is omitted, information is retrieved from the current window.

2. The LINE NUMBER and COLUMN NUMBER phrases return the position of the window relative to the interior of its parent. For the initial window, this will always be line “1”, column “1”. Note that the position returned can be negative, or larger than the parent window, indicating that the window’s upper left corner is outside of its parent’s interior.

3. The TITLE phrase returns the title of the current window.

4. The SCREEN LINE and SCREEN COLUMN phrases return the position of the window on the screen. This is an absolute position, not relative to any other window. The position is expressed in the screen’s base units, with “1, 1” being the upper left corner of the screen. Screen base units are machine dependent. Under character systems, the base unit is a character cell, for graphical systems, the base unit is a pixel. A negative value indicates that the window’s home location (the upper left corner) is off the screen.

5. The SIZE and LINES phrases return the window’s width and height, respectively.

6. The SYSTEM HANDLE phrase retrieves the host graphical system’s handle that corresponds to the window and stores this value in *system-handle*. This value is the way the host graphical system identifies the window. You usually need it if you want to affect the window from some other language such as C. There is no use for the host system’s handle if you are using only ACUCOBOL-GT; the handle is useful only when you need to have another language interact with an ACUCOBOL-GT screen.

7. Each host system defines its own technique for identifying graphical components. Under Windows, the Windows API uses the “HWND” type, which is a 32-bit unsigned value. You can use UNSIGNED-INT as an appropriate USAGE type for *system-handle* to cover these two cases portably.

8. If the window does not have a corresponding host handle, then *system-handle* is set to zero. This indicates that either the host system does not have an underlying graphical system, or that the particular window does not use the host’s notion of a control in its implementation.

9. The LAYOUT-MANAGER phrase sets *layout-manager* to the handle of the layout manager currently attached to the window.

10. The VISIBLE option returns in *visible-state* whether the window is visible or invisible. *Visible-state* is set to "1" if the window is visible, or "0" if it is invisible.

    Note: A "visible" window may actually be hidden behind other windows on the screen, and thus may not actually be viewable by the user.

11. The POP-UP MENU option returns the handle of the pop-up menu associated with the window in *menu-1*. If the window has no pop-up menu, *menu-1* is set to NULL.

INSPECT Statement

The INSPECT statement supports counting and modification of single characters or groups of characters within a data item. INSPECT performs its operations on strings and requires that the source data item be designated USAGE DISPLAY.

INSPECT performs three basic functions: **TALLYING**, **REPLACING**, and **CONVERTING**. (CONVERTING is a specialized, shorthand form of REPLACING).

Note: This manual entry includes code examples and highlights for first-time users following the General Rules section.

General Format

Format 1

```
INSPECT source

 [ TALLYING { counter FOR { { {ALL    } comp-val }
                          { {LEADING} }
                          {     CHARACTERS     }
```

```
            [delim] } ... } ... ] ...

 [ REPLACING

    { CHARACTERS BY repl-char [delim]        }
    { {ALL    }                              }
    { {LEADING} { targ BY repl [delim] } ... }
    { {FIRST  }                              }
                       ... ]
```

Format 2

```
INSPECT source
    CONVERTING comp-chars TO conv-chars  [delim]
```

delim has the following format:

```
    { {BEFORE} INITIAL delimiter } ...
      {AFTER }
```

Format 3

```
INSPECT source

    [ TALLYING counter FOR TRAILING comp-value ]
    [ REPLACING TRAILING target BY replace ]
```

Syntax Rules

1. *Source* is a data item with USAGE DISPLAY.
2. *Counter* is an elementary numeric data item.
3. *Comp-val* is a nonnumeric literal (other than an ALL literal) or an elementary alphabetic, alphanumeric, or numeric data item with USAGE DISPLAY.
4. *Repl-char* is a one-character item with the same restrictions as *comp-val*.
5. *Targ*, *delimiter*, *repl*, *comp-chars*, and *conv-chars* have the same restrictions as *comp-val*.
6. In Formats 1 and 3, at least one of the TALLYING or REPLACING phrases must be specified.

7. Any *delim* phrase may have no more than one AFTER and one BEFORE phrase in it.

8. The sizes of the data referred to by *targ* and *repl* must be the same. If *repl* is a figurative constant, its size is set equal to the size of *targ*.

9. When the CHARACTERS phrase of the REPLACING clause is used, *delimiter* (if specified) must have a data size of one character.

10. The sizes of the data referred to by *comp-chars* and *conv-chars* must be the same. When *conv-chars* is a figurative constant, its size equals that of *comp-chars*.

11. The same character cannot appear more than once in the data referred to by *comp-chars*.

12. *Comp-value*, *target*, and *replace* are nonnumeric literals or single-character alphanumeric data items.

General Rules

1. Inspection starts at the leftmost character of *source* and proceeds character by character until it reaches the rightmost character.

2. *Source*, *comp-val*, *delimiter*, *targ*, *repl*, *repl-char*, *comp-chars*, and *conv-chars* are treated as if they were redefined by an alphanumeric elementary data item. The data referred to by these items is treated as a character string.

3. If the size of *source* is zero characters, no inspection occurs.

4. If the size of *comp-val* or *targ* is zero characters, no match in *source* by these items is successful.

5. *Comp-val* and *targ* are matched in the source string according to the following rules:

    a. Comparison starts at the leftmost character and proceeds character by character until the rightmost character of *source* is reached.

    b. The first *comp-val* or *targ* item is checked at the current character location for a match. A match occurs if every character of *comp-val* or *targ* is the same as the corresponding characters in *source* starting at the current character position.

c. If no match occurs, successive *comp-val* or *targ* items are checked at the current character position until a match occurs or the list of items is exhausted. The next character position is then checked and the process repeats.

d. When a match occurs, the specified tallying or replacement is performed. Further checking for matching items at this character position is not performed. The new next character position to use for matching is set to be the character to the immediate right of the rightmost character position that matched in the preceding comparison.

e. Inspection halts when the rightmost character of *source* has served as the current matching character position or has been successfully matched in the preceding rule.

f. When the CHARACTERS phrase is present, inspection proceeds as if a single-character value were being compared and it successfully matches every character in *source*.

6. The BEFORE phrase modifies the character position to use as the rightmost position in *source* for the corresponding comparison operation. Comparisons in *source* occur only to the left of the first occurrence of *delimiter*. If *delimiter* is not present in *source*, then the comparison proceeds as if there were no BEFORE phrase.

7. The AFTER phrase modifies the character position to use as the leftmost position in *source* for the corresponding comparison operation. Comparisons in *source* occur only to the right of the first occurrence of *delimiter*. This character position is the one immediately to the right of the rightmost character of the *delimiter* found. If *delimiter* is not found in *source*, the INSPECT statement has no effect (no tallying or replacement occurs).

8. If both the TALLYING and REPLACING phrases are present, the TALLYING option is performed first, and then the REPLACING option is performed as if it were written as a separate INSPECT statement.

TALLYING Option

1. The INSPECT statement does not initialize *counter*.

2. If the ALL phrase is present, *counter* is incremented by one for each occurrence of *comp-val* in *source*.

3. If the LEADING phrase is present, *counter* is incremented by one for each contiguous occurrence of *comp-val* in *source*. These occurrences must start at the position in *source* where comparison begins. Otherwise, no tallying occurs.

4. If the CHARACTERS phrase is present, *counter* is incremented by one for each character in *source* that is matched (see General Rule 5f above).

5. If the FOR TRAILING phrase is present, *counter* is incremented by one for each contiguous occurrence of *comp-value* in *source*, starting at the rightmost (trailing) character and scanning leftwards. If the rightmost character is not *comp-value*, then *counter* is not incremented.

REPLACING Option

1. The adjectives ALL, LEADING, and FIRST apply to succeeding compare items until the next such adjective appears.

2. If the CHARACTERS phrase is used, each character matched in *source* is replaced by the single character *repl-char.*

3. When the ALL phrase is present, each occurrence of *targ* matched in *source* is replaced by *repl*.

4. If the LEADING phrase is present, each contiguous occurrence of *targ* matched in *source* is replaced by *repl*. These occurrences must begin at the leftmost position in *source* used for comparison.

5. When the FIRST phrase is present, the leftmost occurrence of *targ* matched in *source* is replaced by *repl*.

6. If the TRAILING phrase is present, the REPLACING option causes all contiguous occurrences of *target* to be replaced by *replace*, provided that these occurrences end in the rightmost character position of *source*.

7. It is possible for a size mismatch between the INSPECT and REPLACING data items to occur during program execution. This could happen when reference modification is used, because in that case the length of a data item is not known at compile time. If such a

mismatch occurs, the runtime generates the "INSPECT REPLACING size mismatch" error. This error belongs to the "intermediate" class of runtime errors which call installed error procedures. See Book 4, *Appendices*, Appendix I "Library Routines," CBL_ERROR_PROC for details.

CONVERTING Option

1. The CONVERTING form of the INSPECT statement has the effect of replacing every character in *source* found in *comp-chars* with the corresponding character in *conv-chars*. This is done according to the following rules:

   a. The INSPECT statement is treated as if it were specified with the REPLACING option containing a series of ALL phrases, one for each character of *comp-chars*.

   b. The *targ* item in each ALL phrase refers to a single character of *comp-chars*.

   c. The *repl* item in each ALL phrase refers to a single character of *conv-chars*.

   d. The individual characters of *comp-chars* and *conv-chars* correspond by ordinal position.

2. INSPECT CONVERTING is usually more efficient than the corresponding INSPECT REPLACING statement.

Code Examples

Example 1:

Use INSPECT to count the number of occurrences of a character or string:

```
01  CHAR-COUNT  PIC 99 VALUE 0.

*count all "b"s
INSPECT INPUT-ITEM
   TALLYING CHAR-COUNT FOR ALL "B".
```

Value of INPUT-ITEM	CHAR-COUNT
#BB44@#AL23#AL88#xx#CC12	2

Value of INPUT-ITEM	CHAR-COUNT
#BB@#BBBB#CCCC#xxDD	6
BB@#BB#BB	6

```
*count all "#"s found after the first "@"
*and before the first "x"
INSPECT INPUT-ITEM
  TALLYING CHAR-COUNT FOR ALL "#":
     AFTER "@" BEFORE "x".
```

Value of INPUT-ITEM	CHAR-COUNT
#BB44@#AL23#AL88#xx#CC12	3
#BB@#BBBB#CCCC#xxDD	3
BB@#BB#BB	2

```
*count all characters
INSPECT INPUT-ITEM
   TALLYING CHAR-COUNT FOR CHARACTERS.
```

Value of INPUT-ITEM	CHAR-COUNT
#BB44@#AL23#AL88#xx#CC12	24
#BB@#BBBB#CCCC#xxDD	19
BB@#BB#BB	9

Example 2:

Use INSPECT to replace matching characters or strings:

```
INSPECT NAME-LIST REPLACING
*if the first characters in the string are "a"
*replace the "a"s with "A"s
   LEADING "a" BY "A"
*replace all "T"s found after the first "/"
*with "t"
   ALL "T" BY "t" AFTER "/"
*replace all "/"s with ":"
   ALL "/" BY ":"
*after the first "-" replace all characters
*in the string with "Z"
```

```
   CHARACTERS BY "Z" AFTER "-".
```

Input value NAME-LIST	Output value NAME-LIST
TED/TRAVIS/UREY/VENNEY	TED:tRAVIS:UREY:VENNE Y
aVERY/BLAZE/TERI	AVERY:BLAZE:tERI
MAVIS-GUS-HAL-WESTON	MAVIS-ZZZZZZZZZZZZZZ

Example 3:

Use INSPECT to both tally and replace characters or strings:

```
INSPECT PART-LIST
*count all "P-"
   TALLYING P-COUNT FOR ALL "P-"
*replace all "xx" by "__"
   REPLACING ALL "xx" BY "__".
```

Value of PART-LIST	P-COUNT
P-BOLTxxP-WASHERxxP-NUT	3

Input value PART-LIST	Output value PART-LIST
P-BOLTxxP-WASHERxxP-NUT	P-BOLT__P-WASHER__P-NUT

Example 4:

Use INSPECT/CONVERT to convert every occurrence of the specified characters in the input string (equivalent to a series of REPLACE ALL phrases). The list of characters following the words CONVERT and TO is not a string, but, rather, a list of individual characters. INSPECT/CONVERT replaces every occurrence of each character in the CONVERT list with the character in the matching ordinal position of the TO list. AFTER and BEFORE can be used to bracket a portion of the source string.

```
*convert all occurrences of:
*"-" to "0", "l" to "L",
*"a" to "A" and "/" to ":"
INSPECT PART-LIST
   CONVERTING "-la/" TO "0LA:".
```

This INSPECT/CONVERT statement is equivalent to:

```
INSPECT PART-LIST
   REPLACING ALL "-" BY "0"
             ALL "l" BY "L"
             ALL "a" BY "A"
             ALL "/" BY ":".
```

Input value PART-LIST	Output value PART-LIST
Yla-1/Yla-2/Yla-3/Yla21	YLA01:YLA02:YLA03:YLA21

Highlights for first-time users

1. How the matching process works:

   In all formats of the INSPECT statement there must be a set of specified "match" values. The match values may be single characters or strings, or a mix of both. The match values are the arguments specified after the TALLYING/FOR, REPLACING, or CONVERTING key words, and before the BY or TO key words (for further clarification see the example that follows).

   INSPECT attempts to locate the match values in the source data item. The match values are searched for, in order of appearance in the code, at each position in the source string. Inspection starts at the leftmost character of the source data item and proceeds, character by character, to the rightmost character.

   Match process example:

   ```
   INSPECT source-data-item
     REPLACING "cd" BY "QP"
               "e"  BY "T"
               "f"  BY "V".
   ```

   Source data item: "abcdefg"

   Set of match values: "cd", "e", "f"

   The search begins at the leftmost character:

   ```
   "abcdefg"
    ^
   ```

The first value in the match set is “cd”. The first element, “c”, is compared to the value “a” for a match. No match. The second value in the match set is “e”. “e” is tested for a match with “a”. No match. The third value in the match set is “f”. “f” is tested for a match with “a”. No match. There are no untested values remaining in the match set. The current position in the source data item is advanced one position to the right.

```
"abcdefg"
  ^
```

The sequential testing of the members of the match set to the value of the current position in the source data item is repeated. None matches. The current position in the source data item is advanced one position to the right.

```
"abcdefg"
   ^
```

The first match value is “cd”. “c” is tested for a match with the current position in the source data item and there is a match. “d” is tested against the value of 1 + the current position (“d”) and, again, there is a match. There are no more characters in the match value (“cd”). The first value in the match set results in a complete match at the current position in the source data item. Remaining values in the match set are not tested. The specified REPLACING action is performed. The current position in the source data item is advanced the length of the match value (“cd”), two places to the right.

```
"abQPefg"
     ^
```

Sequential testing of the members of the match set to the value of the current position in the source data item is repeated. The second member of the match set, “e”, is a match. The REPLACING action is performed and the current position in the source data item is advanced the length of the match value, one position to the right.

```
"abQPTfg"
      ^
```

Notice that, if we had replaced “cd” with “ef” instead of “QP”, the “e” and “f” would *not* be subsequently replaced with “T and “V”.

Sequential testing of the members of the match set to the value of the current position in the source data item is repeated. The third member of the match set, “f”, is a match. The REPLACING action is performed and the current position in the source data item is advanced the length of the match value, one position to the right.

```
"abQPTVg"
        ^
```

Sequential testing of the members of the match set to the value of the current position in the source data item is repeated. None matches. The current position in the source data item is the rightmost character; therefore, the inspection halts.

For a more concise description of the matching process, see General Rule 5 above.

2. All replacement actions must replace the same number of characters as matched. The source data item may not change in size.
3. Use AFTER and BEFORE to bracket (define) a substring in the source data item.
4. Use ALL to match all occurrences of the specified value in the string.
5. Use CHARACTERS to match every character in the source data item that hasn’t already been matched. Because CHARACTERS matches all elements of the source data item, CHARACTERS usually appears as the last phrase in the statement.
6. Use LEADING to find the leading occurrence, or set of leading contiguous occurrences, in the source data item.
7. Use TRAILING (ACUCOBOL-GT extension) to find the rightmost occurrence, or set of contiguous occurrences, in the source data item. If a TRAILING occurrence is found, a right to left scan of the source data item is made to find contiguous occurrences.
8. Use FIRST to specify a match of the first occurrence only.
9. Many COBOL programming texts caution against writing involved and complicated INSPECT statements, because complex statements are difficult to understand and maintain.

LOCK Statement

The LOCK THREAD statement prevents other threads from running.

General Format

```
LOCK THREAD
```

General Rules

1. The LOCK THREAD statement prevents other threads from running. The thread that executes the LOCK statement is the only thread allowed to run until an UNLOCK THREAD statement is executed, or the thread terminates. Locking a thread ensures that other threads will not modify a critical piece of data or other shared resource.

2. A thread can be locked multiple times. Each time a thread executes a LOCK THREAD statement, the number of locks held by the thread increases by one. In order to unlock a thread with multiple locks, an equal number of UNLOCK THREAD statements must execute.

   This allows a thread to lock itself, call a subroutine that also locks itself, and remain locked when that subroutines unlocks itself. See UNLOCK THREAD.

MERGE Statement

The MERGE statement combines two or more identically ordered files by selected ASCENDING or DESCENDING key fields.

Unlike SORT, MERGE doesn't allow you to manipulate the records before they are merged. Like SORT, MERGE does allow you to modify records after they are merged via the OUTPUT PROCEDURE phrase.

Note: This manual entry includes code examples and highlights for first-time users following the General Rules section.

General Format

```
MERGE merge-file

    { KEY AREA IS key-table       }

    { ON {ASCENDING } KEY {key-name}  } ...
         {DESCENDING}

    [ COLLATING SEQUENCE IS alpha-name ]

      USING {in-file} ...

    { OUTPUT PROCEDURE IS proc-name  }
    { GIVING {out-file} ...          }
```

proc-name has the following format:

```
start-proc [ {THRU   } end-proc ]
             {THROUGH}
```

Syntax Rules

1. *Merge-file* names a sort file described by an SD entry in the Data Division.

2. *Key-table* must name a data item that is *not* located in the record for merge-file. *Key-table* may not be subordinate to an OCCURS clause, nor may it be reference modified.

3. *Key-table* must reference a data item whose size is an even multiple of 7. *Key-table* is processed as if it had the following structure:

```
01  KEY-TABLE.
    03  MERGE-KEY OCCURS N TIMES.
        05  KEY-ASCENDING   PIC X  COMP-X.
        05  KEY-TYPE        PIC X  COMP-X.
        05  KEY-OFFSET      PIC XX COMP-X.
        05  KEY-SIZE        PIC XX COMP-X.
        05  KEY-DIGITS      PIC X  COMP-X.
```

   Typically, programs will declare *key-table* with a similar format.

4. *Key-name* is a data item in the record description associated with *merge-file*. It may not be subordinate to an OCCURS clause, nor may it be a group item containing variable occurrence data items. The maximum number of keys allowed is 23.

5. *Alpha-name* is an alphabet-name defined in the SPECIAL-NAMES paragraph of the Environment Division.

6. *In-file* and *out-file* are files described by FD entries in the Data Division. They may not be sort files. The maximum number of input and output files allowed is 25.

7. *Start-proc* and *end-proc* are paragraph or section names in the Procedure Division.

8. A MERGE statement may not appear in Declaratives or in the input or output procedure of a SORT or MERGE statement.

9. If *merge-file* contains variable length records, *in-file* records must not be smaller than the smallest record in *merge-file* nor larger than the largest. If *merge-file* contains fixed length records, *in-file* records may not be larger than the size of *merge-file's* records.

10. If *out-file* contains variable length records, *merge-file* records must not be smaller than the smallest record in *out-file* nor larger than the largest. If *out-file* contains fixed length records, *merge-file* records may not be larger than the size of *out-file's* records.

11. If *merge-file* contains more than one record description, *key-name* need appear in only one of them. The character positions referenced by *key-name* are used as the key for all the file's records.

12. If *out-file* is an indexed file, the first *key-name* must be ASCENDING and must specify the same character positions in its record as the primary record key for *out-file*.

13. THRU is an abbreviation for THROUGH.

General Rules

1. The MERGE statement merges all the records in the *in-file* files into *merge-file* and then either writes these records to each *out-file* or makes these records available to the specified OUTPUT PROCEDURE.

2. If *merge-file* contains fixed length records, any shorter *in-file* records are space-filled on the right to match the record size.

3. If *out-file* contains fixed length records, any shorter *merge-file* records are space-filled on the right to match the record size.

4. The first *key-name* is the major key, and the next *key-name* is the next most significant key. This pattern continues for each *key-name* specified.

5. The ASCENDING phrase specifies that key values are to be ordered from lowest to highest. The DESCENDING phrase specifies the reverse ordering. Once ASCENDING or DESCENDING is specified, it applies to each *key-name* until another ASCENDING or DESCENDING adjective is encountered.

6. Use the KEY AREA option when you do not know the specifics of the merge key until the program is run. You can use this to allow users to enter merge key specifications, typically in conjunction with some form of data dictionary.

7. Your program must fill in a table of information that describes the merge keys. This table, *key-table*, should have the format described by Syntax Rule 3 above. The number of merge keys is determined by the number of occurrences in the table. The keys are listed in order of precedence: table entry 1 describes the highest precedence key, table entry 2 the second highest, and so on. If you need to process a variable number of keys, use a variable-size table (by using OCCURS DEPENDING ON).

8. For each key, you must specify the following information:

KEY-ASCENDING:	This should be 0 or 1. Enter 1 to have an ascending merge sequence, 0 for descending.
KEY-TYPE:	Describes the underlying data format. The allowed values are listed in the next rule.
KEY-OFFSET:	Describes the distance (in standard character positions) from the beginning of the merge record to the beginning of the key field. The first field in a merge record is at offset 0.

KEY-SIZE:	Describes the size of the key field in standard character positions.
KEY-DIGITS:	This is used only for numeric keys. It describes the number of digits contained in the key (counting digits on both sides of the decimal point).

9. The KEY-TYPE field uses a code to describe the type and internal storage format of the data item. Select from the following values:

0	Numeric edited
1	Unsigned numeric (DISPLAY)
2	Signed numeric (DISPLAY, trailing separate)
3	Signed numeric (DISPLAY, trailing combined)
4	Signed numeric (DISPLAY, leading separate)
5	Signed numeric (DISPLAY, leading combined)
6	Signed COMP-2
7	Unsigned COMP-2
8	Unsigned COMP-3
9	Signed COMP-3
10	COMP-6
11	Signed binary (COMP-1, COMP-4, COMP-X)
12	Unsigned binary (COMP-1, COMP-4, COMP-X)
13	Signed native (COMP-5, COMP-N)
14	Unsigned native (COMP-5, COMP-N)
15	Floating point (FLOAT, DOUBLE)
16	Alphanumeric
17	Alphanumeric (justified)
18	Alphabetic
19	Alphabetic (justified)
20	Alphanumeric edited
22	Group

This coding is the same one used by the C interface, and is also used by Acu4GL. When specifying the key type, you may safely use "alphanumeric" for all nonnumeric keys. (The merge rules are the same for each of these types). For numeric data, however, you must specify the correct type or you may get merging errors.

10. The results are undefined if you provide invalid data in the *key-table*. If you fail to specify any keys (by specifying a table whose size is zero), you receive a file error on *merge-file*. Under the default file status codes, this is file error 94 with a secondary status of 63.

11. For nonnumeric keys, the COLLATING SEQUENCE phrase establishes the ordering. If this phrase is omitted, the NATIVE collating sequence is used. For numeric keys, the ordering is specified by the algebraic value of the key.

12. When the contents of all key fields in one input record equal the contents of the key fields in another, the order of return:

    a. follows the order of the associated *in-files* in the MERGE statement

    b. causes all records with equal key values from one input file to be returned before any are returned from another

13. The MERGE statement transfers all records from each *in-file* to *merge-file*. When the MERGE statement executes, *in-file* must not be open. The results of the MERGE statement are undefined if the *in-file* records are not ordered according to the KEY clause of the MERGE statement.

14. For each *in-file*, the MERGE statement:

    a. opens the file as if it had been the object of an OPEN INPUT statement with no options. This occurs before any associated output procedure executes.

    b. retrieves the records of the file and releases them to the merge operation. The retrieval is performed as if the program had executed a READ statement with the NEXT and AT END phrases.

    c. closes the file as if it were the object of a CLOSE statement with no options. This occurs after any associated output procedure has finished execution.

These actions cause any associated USE procedures to execute if an exception condition occurs.

15. The OUTPUT PROCEDURE, if specified, is executed by the MERGE statement when the records are ready to be processed in merged order. The statements in the range of the output procedure must contain one or more RETURN statements to retrieve the merged records. Control is passed to the output procedure by the MERGE statement according to the rules of the PERFORM statement. When the last statement of the output procedure is executed, control returns to the MERGE statement. The MERGE statement then closes the *in-files* and terminates.

16. If the MERGE statement is in a fixed segment, the range of the output procedure must be contained completely in the fixed segments and no more than one independent segment. If the MERGE statement is in an independent segment, the range must be completely contained in the fixed segments and the same independent segment.

17. If the GIVING phrase is used, the MERGE statement writes all merged records to each *out-file*. *Out-file* must not be open when the MERGE statement executes.

18. The MERGE statement writes records to *out-file* with the following steps:

    a. *Out-file* is opened as if it were the object of an OPEN OUTPUT statement with no options.

    b. Each merged record is retrieved and written to *out-file* as if it were the object of a WRITE statement.

    c. *Out-file* is closed as if it were the object of a CLOSE statement with no options.

19. The implicit OPEN, WRITE, and CLOSE operations cause associated USE procedures to execute if an exception condition occurs. If the MERGE statement tries to write beyond the boundaries of *out-file*, the applicable USE procedure executes. If that procedure returns, or no USE procedure is specified, the processing of that *out-file* terminates with an implied CLOSE operation.

20. If *out-file* is a relative file, the value of the RELATIVE KEY data item is updated to contain the record number of each record after it is written.

21. The MERGE statement updates the value of the FILE STATUS data item associated with *merge-file*.

22. If a MERGE statement is executed in a wrong context, the runtime displays the error "Illegal MERGE." This error belongs to the class of "intermediate" runtime errors that, upon occurrence, call installed error procedures. See Book 4, *Appendices*, Appendix I "Library Routines," CBL_ERROR_PROC for details.

Code examples

Example 1:

```
*Merge sales prospects lists.
   MERGE NATIONAL-MERGE-FILE
      ON ASCENDING KEY PROSPECT-CLASS
                       SALES-REP-NUMBER
      USING  WESTERN-REGION-FILE,
             EASTERN-REGION-FILE,
             SOUTHERN-REGION-FILE
      GIVING NATIONAL-PROSPECT-FILE.
```

Example 2:

(An extended code sample of this example may be found at the end of this reference entry.)

```
*Merge sales prospects lists and use an
*OUTPUT PROCEDURE to do processing on the list
*before writing it to the output file.
   MERGE NATIONAL-MERGE-FILE
      ON ASCENDING KEY PROSPECT-CLASS
                       SALES-REP-NUMBER
      USING  WESTERN-REGION-FILE,
             EASTERN-REGION-FILE,
             SOUTHERN-REGION-FILE
      OUTPUT PROCEDURE IS PROCESS-PROSPECT-LIST.
```

Highlights for first-time users

1. MERGE can be thought of as a specialized version of SORT that has been optimized to give better processing performance than can be achieved using SORT. Bear in mind, however, that MERGE, like SORT, does all of its I/O on disk files and will, therefore, take a variable amount of time to complete, depending on the size of the input files, the number of records in the files and the speed of the disk subsystems.

2. MERGE does not allow the use of an input procedure for manipulating records before they are merged.

3. The files to be merged must have identical record formats and be identically ordered by the same key fields.

4. The result of the merge may be written directly to an output file or made available to an output procedure.

5. The output procedure may not reference any of the input files or their records. You can access the records contained in the input files, in merged order, by using RETURN to fetch records from the merge file.

6. The KEY AREA phrase is a means for defining the merge keys at runtime. When you use KEY AREA, it is not required that the merge file record descriptor contain entries for potential sort keys. Definition of the sort key(s) in the merge file is handled internally by the MERGE routine, using the key table. See syntax rules 2 and 3 and general rules 6 through 10.

7. Summary of the merge process:

    a. At the beginning of the MERGE process all input files (in-files) and the temporary merge file (merge-file) are opened and positioned at the head of the file. The input files cannot already be open when the MERGE statement begins.

    b. The records of each input file are sequentially READ and released to the merge operation.

    c. When all of the records in all of the input files have been read, the input files are closed and MERGE completes its merging process.

d. Following merge processing, if OUTPUT PROCEDURE is specified, control is passed to the output procedure. In the output procedure, each record in the merge file is fetched, in sort order, by the RETURN verb for processing (see the entry for the RETURN statement in this section). When the last statement of the output procedure is executed, control returns to the MERGE statement.

e. If the GIVING phrase is used, the merged records are written to the specified output file(s).

Extended code example 2:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  SAMPLE-FILE-MERGE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT WESTERN-REGION-FILE
      ASSIGN TO ....
   SELECT EASTERN-REGION-FILE
      ASSIGN TO ....
   SELECT SOUTHERN-REGION-FILE
      ASSIGN TO ....
   SELECT NATIONAL-PROSPECT-FILE
      ASSIGN TO ....
   SELECT NATIONAL-MERGE-FILE
      ASSIGN TO ....
DATA DIVISION.
FILE SECTION.
FD  WESTERN-REGION-FILE.
01  W-REGION-RECORD        PIC X(30).
FD  EASTERN-REGION-FILE.
01  E-REGION-RECORD        PIC X(30).

FD  SOUTHERN-REGION-FILE.

01  S-REGION-RECORD        PIC X(30).

SD  NATIONAL-MERGE-FILE.

01  SORT-DATA.
    05  PROSPECT-NUMBER    PIC X(5).
    05  PROSPECT-NAME      PIC X(7).
    05  PROSPECT-CLASS     PIC X.
```

```
    05  ESTIMATED-VALUE    PIC 9999V9.
    05  SALES-REP-NUMBER   PIC X(3).
    05  SALES-REP-NAME     PIC X(7).
    05  FILLER             PIC XX.
FD  NATIONAL-PROSPECT-FILE.
01  NATIONAL-RECORD        PIC X(30).
WORKING-STORAGE SECTION.
01  FLAGS.
    05   MERGE-LIST-EMPTY   PIC X VALUE "N".
         88 NO-MORE-RECORDS       VALUE "Y".
...
PROCEDURE DIVISION.
PROSPECT-LIST-MERGE-PROCEDURE.
    MERGE NATIONAL-MERGE-FILE
       ON ASCENDING KEY PROSPECT-CLASS
                        SALES-REP-NUMBER
       USING  WESTERN-REGION-FILE,
              EASTERN-REGION-FILE,
              SOUTHERN-REGION-FILE
       OUTPUT PROCEDURE IS PROCESS-PROSPECT-LIST.

PROCESS-PROSPECT-LIST SECTION.
CREATE-NATIONAL-PROSPECT-FILE.
   OPEN OUTPUT NATIONAL-PROSPECT-FILE.
   RETURN NATIONAL-MERGE-FILE
      AT END MOVE “Y” TO MERGE-LIST-EMPTY.
   PERFORM UPDATE-PROSPECT-DATA
      UNTIL NO-MORE-RECORDS.
   CLOSE NATIONAL-PROSPECT-FILE.
   GO TO EXIT-MERGE-OUTPUT-PROCESSING.

UPDATE-PROSPECT-DATA.
*do not write records tagged "TestRep"
   IF SALES-REP-NAME NOT = "TestRep"
*write the record to the output file
      WRITE NATIONAL-RECORD FROM SORT-DATA.
   END-IF.
*fetch the next record
   RETURN NATIONAL-MERGE-FILE
      AT END MOVE “Y” TO MERGE-LIST-EMPTY.

EXIT-MERGE-OUTPUT-PROCESSING.
   EXIT.
```

MODIFY Statement

The MODIFY verb is used to change the characteristics of an existing screen control item or window. It acts on control handles, elementary Screen Section control items, and window handles.

General Format

Format 1

```
MODIFY {control-item} [ ( {index-1} ... ) ]
       {CONTROL     }
```

Remaining phrases are optional, can appear in any order.

```
AT screen-loc   [CELL  ]
                [CELLS ]
                [PIXEL ]
                [PIXELS]

AT LINE NUMBER line-num   [CELL  ]
                          [CELLS ]
                          [PIXEL ]
                          [PIXELS]

AT {COLUMN  } NUMBER col-num   [CELL  ]
   {COL     }                  [CELLS ]
   {POSITION}                  [PIXEL ]
   {POS     }                  [PIXELS]

AT CLINE NUMBER cline-num [CELL ]
                          [CELLS]

AT CCOL NUMBER ccol-num [CELL ]
                        [CELLS]

SIZE    {IS} length  [CELL  ]
        {= }         [CELLS ]
                     [PIXEL ]
                     [PIXELS]

LINES   {IS} height  [CELL  ]
        {= }         [CELLS ]
                     [PIXEL ]
```

```
                        [PIXELS]

CSIZE   {IS} clength  [CELL ]
        {= }          [CELLS]

CLINES  {IS} cheight  [CELL ]
        {= }          [CELLS]

MAX-HEIGHT {IS} max-height
           {= }

MAX-WIDTH  {IS} max-width
           {= }

MIN-HEIGHT {IS} min-height
           {= }

MIN-WIDTH  {IS} min-width
           {= }

TITLE {IS} title
      {= }

{COLOR } IS color-val
{COLOUR}

{FOREGROUND-COLOR } IS fg-color
{FOREGROUND-COLOUR}

{BACKGROUND-COLOR } IS bg-color
{BACKGROUND-COLOUR}

{HIGHLIGHT}
{HIGH     }
{BOLD     }
{LOWLIGHT }
{LOW      }
{STANDARD }

{BACKGROUND-HIGH}
{BACKGROUND-LOW}
{BACKGROUND-STANDARD}

STYLE {IS} style-flags
      {= }
```

```
{ [NOT] style-name } ...

VALUE {IS} [ MULTIPLE ] value [ LENGTH {IS} length-1 ]
      {= } [ TABLE    ]               {= }

LAYOUT-DATA {IS} layout-data
            {= }

FONT  {IS} font-handle
      {= }

ENABLED  {IS} {TRUE         }
         {= } {FALSE        }
              {enabled-state}

VISIBLE  {IS} {TRUE         }
         {= } {FALSE        }
              {visible-state}

POP-UP MENU {IS} {menu-1}
            {= } {NULL  }

EVENT-LIST    {IS} ( event-value { event-value ... } )
              {= }

AX-EVENT-LIST {IS} ( ax-event-value { ax-event-value ... } )
              {= }

EXCLUDE-EVENT-LIST {IS} list-state
                   {= }

EVENT PROCEDURE IS { proc-1 [ {THROUGH} proc-2 ] }
                             {THRU   }
                   { NULL                      }

{ property-name          } {IS } { prop-option
                                       [GIVING result-1] }...
{ PROPERTY property-type } {ARE}
{ method-name            } {=  }
{ object-expression      }
```

prop-option is one of the following:

```
{ property-value [ LENGTH {IS} length-1 ] }
```

```
{                          {= }           }
{                                         )
{ ( {property-value} ... )                }
{                                         }
{ { MULTIPLE } property-table             }
{ { TABLE    }                            }
{                                         }
{ parameter-expression                    }
{                                         }
{ ( { parameter-expression } ... )        }
```

parameter-expression is one of the following:

```
{ parameter                                 } [ AS type-num ]
{                                           }
{ {BY} NAME parameter-name {IS} parameter   }
{                           {= }            }
{ parameter-name {IS} parameter             }
{                {= }                       }
```

object-expression has the following format:

```
{ {^} property-1 [ (param-expr ... ) ]
  [ :: property-2 [ ( param-expr ... ) ] ... }
```

Format 2

```
MODIFY {window-handle           }
       {WINDOW [generic-handle] }
```

Remaining phrases are optional, can appear in any order.

```
AT screen-loc

LINE NUMBER line-num

{COLUMN  } NUMBER col-num
{COL     }
{POSITION}
{POS     }

SCREEN LINE NUMBER screen-line

SCREEN {COLUMN  } NUMBER screen-col
       {COL     }
       {POSITION}
```

```
       {POS     }

SIZE width

LINES height

TITLE title

ON EXCEPTION statement-1

NOT ON EXCEPTION statement-2

LAYOUT-MANAGER {IS} manager
               {= }

VISIBLE  {IS} {TRUE         }
         {= } {FALSE        }
              {visible-state}

POP-UP MENU {IS} {menu-1}
            {= } {NULL  }

ENABLED  {IS} {TRUE         }
         {= } {FALSE        }
              {enabled-state}

EVENT PROCEDURE IS { proc-1 [ {THROUGH} proc-2 ] }
                                {THRU   }
                   { NULL                       }

ACTION  {IS} action
        {= }

END-MODIFY
```

Syntax Rules

1. *Control-item* is a USAGE HANDLE data item or elementary Screen Section item that describes a control.

2. *Index-1* is a numeric expression. The parentheses surrounding *index-1* are required.

3. *Window-handle* is a USAGE HANDLE OF WINDOW or PIC X(10) data item.

4. *Generic-handle* is a USAGE HANDLE, HANDLE OF WINDOW or PIC X(10) data item.

5. *Screen-loc* is an integer data item or literal that contains exactly 4, 6, or 8 digits, or a group item of 4, 6, or 8 characters.

6. *Line-num, col-num, cline-num, ccol-num, length, height, width, clength,* and *cheight* are numeric data items or literals. They can be non-integer values, except when pixels are specified.

7. *Screen-line* and *screen-col* are numeric expressions. They should be integer values.

8. If the CELLS option is used with either the SIZE or CSIZE phrase, then it must be present in both phrases if both are specified. The same is true for use of the CELLS option in the LINES and CLINES phrases.

9. *Max-height, max-width, min-height,* and *min-width* are numeric data items, literals, or expressions.

10. *Color-val* is an integer data item or literal.

11. *Fg-color* and *bg-color* are integer literals or numeric data items. They may be arithmetic expressions. See **section 6.4.9**, "FOREGROUND-COLOR and BACKGROUND-COLOR Phrases", for a more detailed discussion of color settings and values.

12. If you use the AT phrase, you may not use the LINE, COLUMN, SCREEN LINE, or SCREEN COLUMN phrases.

13. The SCREEN LINE and SCREEN COLUMN phrases must be used together. If used, the AT, LINE, and COLUMN phrases may not be used.

14. If the COLOR phrase is specified, neither the FOREGROUND-COLOR nor the BACKGROUND-COLOR phrase may be specified.

15. *Style-flags* is a numeric expression.

16. *Style-name* is the name of a style associated with the class of control being described. The *style-name* phrase adds the named style to the control. If *control-handle* refers to a generic handle, or if the CONTROL phrase is used, you may not use the *style-name* phrase. Use the STYLE phrase instead. If the NOT option is used with the *style-name* phrase, the named style is removed from the control instead. When a style is added, any conflicting styles are removed first. For example, if you add the FRAMED style to a button, then the UNFRAMED style is removed first.

17. *Value* is a literal or data item. If the MULTIPLE option is specified, then *value* must be a one-dimensional table. In this case, *value* is not subscripted.

18. *Length-1* is a numeric literal or data item. The LENGTH phrase may be specified only if the *value* or *property-value* immediately preceding it is an alphanumeric literal or data item, and not a figurative constant. In addition, the MULTIPLE option may not be specified along with the LENGTH phrase.

19. *Title* is an alphanumeric literal or data item.

20. *Layout-data* is an integer literal, data item, or expression.

21. *Manager* is a USAGE HANDLE or HANDLE OF LAYOUT-MANAGER data item that contains a valid reference to a layout manager.

22. *Font-handle* is a USAGE HANDLE data item that identifies a font.

23. *Enabled-state* and *visible-state* are integer numeric literals or data items.

24. *Menu-1* is a USAGE HANDLE or HANDLE OF MENU data item.

25. *Event-value* and *ax-event-value* are numeric literals or data items that identify an event type. List elements must be enclosed by parentheses. Elements must be separated by a space. If the list contains a single element, the parentheses can be omitted.

26. *List-state* is an integer literal or numeric data item. Valid values are “0” and “1”.

27. *Proc-1* and *proc-2* are procedure names.

28. You must allow recursive paragraphs in order to specify the EVENT PROCEDURE phrase. Compiling for recursive paragraphs is allowed by default, but you can turn it off if you use the "-Zr0" option.

29. *Property-name* is the name of a property specific to the type of control being referenced. If the type of control is unknown to the compiler (as in a "DISPLAY OBJECT object-1" statement), then *property-name* may not be used. You must use the PROPERTY *property-type* option instead.

30. *Property-type* is a numeric literal or data item. It identifies the property to modify. The numeric values that identify the various control properties can be found in the COPY library "controls.def".

31. *Method-name* is the name of method specific to the type of ActiveX control or COM object being referenced. If the type of the control or object is unknown to the compiler, then *method-name* cannot be used. You must use the PROPERTY property-type option instead.

32. *Property-value* is a literal or data item. In the Procedure Division, *property-value* may also be a numeric expression (however, only the first *property-value* in a phrase may be an expression, subsequent values must be literals or data items). Note that the parentheses are required.

33. *Property-table* is a data item that appears in a one-dimensional table. No index should be specified.

34. *Result-1* is a numeric data item.

35. In *parameter-expression*:

    a. *Parameter* is a literal, data-item, or numeric expression used when invoking methods or setting properties of an ActiveX control or COM object.

    b. *Type-num* is a numeric data item or numeric literal.

36. In *object-expression*:

    a. ^ can only be used in conjunction with a Format 5 USE verb for an ActiveX control or COM object.

    b. *Property-1* is the name of a property of the ActiveX control or COM object. *Property-1* cannot be a write-only property.

c. *Property-2* is the name of a property of the ActiveX control or COM object that is the value of *property-1*. *Property-2* cannot be a write-only property.

37. *Statement-1* and *statement-2* are imperative statements.

38. *Action* is a numeric literal or a data item.

General Rules

Format 1 (MODIFY CONTROL)

1. A Format 1 MODIFY statement updates an existing control or invokes a method on an ActiveX control, COM object, or .NET control (also known as an assembly). *Control-item* should contain a handle returned by a DISPLAY Control-Type statement, or the name of an elementary Screen Section control item. If *control-item* does not refer to a valid control, the MODIFY statement has no effect. Note that controls referenced in the Screen Section are not valid until they have been created via a DISPLAY statement. If *control-item* refers to a valid control, the effect of the statement is to update the specified properties of the control and to redisplay it.

2. If *index-1* is specified, then certain properties in the control being modified are changed to match the value of *index-1*. This occurs before any modification occurs. The exact set of properties changed by the *index-1* depends on the control's type. Currently, two controls have properties that are changed in this way:

Control Type	Properties Affected
List Box	QUERY-INDEX
Grid	Y, X

Each occurrence of *index-1* changes one property. The first occurrence changes the first property in the list presented in the preceding table. The second occurrence changes the second property. For example, the statement fragment

```
MODIFY grid-1(2, 3), color is red
```

would have the effect of setting the grid property “Y” to “2” and “X” to “3” before changing the cell color to red.

Supplying more index values than the control supports has no additional effect. You may omit trailing indexes; this leaves the corresponding properties unchanged.

This feature can be used to simplify modification of specific elements of controls that hold multiple values. For example, you can modify the contents of row 2, column 3 in a grid with the statement:

```
MODIFY grid-1(2, 3), CELL-DATA = data-1
```

This is exactly equivalent to the more cumbersome:

```
MODIFY grid-1, Y = 2, X = 3
MODIFY grid-1, CELL-DATA = data-1
```

3. The meaning of each of the phrases is the same as for a Format 14 DISPLAY statement. Note that you can move a control by changing its row or column property.

4. MODIFY simply locates the corresponding control and makes the specified modifications. This process does not examine any phrases specified in the Screen Section.

   This capability is particularly convenient when you want to make one or two changes to a Screen Section control. For example, if you want to add an item to a list box, you can simply modify the list box specifying the “item-to-add” property. For example:

   ```
   * Screen Section
   01  list-box-1, list-box, value list-item, line 5,
          column 15, size 20, lines 6.

   * Procedure Division
   modify list-box-1, item-to-add = new-list-item.
   ```

   By using the MODIFY verb, you do not need to specify an “item-to-add” property in the Screen Section, and thus you do not need to closely manage the “item-to-add” variable.

5. If the CONTROL phrase is used, the runtime modifies the control located at the screen position specified by the AT, LINE, and COLUMN phrases in the current window (on non-graphical systems,

the CLINE and CCOL phrases also apply). The runtime maintains a list of controls in each window. When attempting to modify a control at a specific location, the runtime searches this list, using the first control it finds that exactly matches the location. The list is maintained in the order in which the controls are created. If the runtime does not find a control at the specified location, then the statement has no effect.

6. Note that you cannot move a control with a MODIFY statement if it includes the CONTROL phrase. This is due to the fact that the AT, LINE, and COLUMN phrases are used to find the control instead of specifying its new position. To move a control, you must use the *control-handle* phrase instead. Also note that when you use the CONTROL phrase, the compiler does not know the type of control being modified. This means that the compiler will not recognize any control-type specific style and property names. If you need to specify these, you will need to use their numeric equivalents found in the "controls.def" COPY library.

   The following example creates an anonymous list box and adds two items to it. Note the use of the PROPERTY phrase in the MODIFY statement: the compiler does not know that the control is a list box so it does not recognize the list-box specific property names. As a result, the generic PROPERTY phrase is used in the example, specifying the level 78 data name that corresponds to the ITEM-TO-ADD property (found in "controls.def").

   ```
   COPY "controls.def".

   DISPLAY LIST-BOX, LINE 5, COL 30, LINES 5.
   MODIFY CONTROL, LINE 5, COL 30,
      PROPERTY LBP-ITEM-TO-ADD =
               ( "Item 1", "Item 2" ).
   ```

7. The *style-name* phrase adds the named style to the control. If the NOT option is used with the *style-name* phrase, the named style is removed from the control instead. When a style is added, any conflicting styles are removed first. For example, if you add the FRAMED style to a button, then the UNFRAMED style is removed first.

8. When the LENGTH option is specified, *length-1* establishes the exact size of the *value* or *property-value*. The text value presented to the control may have no trailing spaces or may have trailing spaces added. When you specify the LENGTH option, the control uses exactly

length-1 characters of data with or without trailing spaces. However, when *length-1* is a value larger than the size of the data item it is modifying, then the size of the data item is used instead. If *length-1* is negative, it is ignored and the default handling occurs.

9. The POP-UP MENU option changes the pop-up menu for the control. If *menu-1* is specified, then the corresponding menu becomes the new pop-up menu. If NULL is specified, any existing pop-up menu is removed (but not destroyed).

10. The EVENT PROCEDURE phrase adds, changes, or removes a control's event procedure. Specifying NULL removes any event procedure. Otherwise, *proc-1* (through *proc-2*, if specified) becomes the control's new event procedure.

11. When properties return specific values, these values are placed in *result-1* of the GIVING phrase. If the property does not have a pre-defined return value, *result-1* is set to "1" if the property is set successfully, otherwise, *result-1* is set to "0". When a property is being given multiple values in a single assignment, as shown here,

    ```
    DISPLAY COLUMNS = ( 1, 10, 30 )
    ```

    then *result-1* is set in response to the last value assigned. In the example above, *result-1* is set to 30. Because the meaning of each value depends on the property being set, you should consult the documentation on the specific property for the exact meaning.

12. You can also change the properties of most controls described in the Screen Section with a Format 2 DISPLAY statement. You must use MODIFY to change special properties of an ActiveX or .NET control.

13. To invoke (call) a method, you use the MODIFY verb in much the same way as you set a property or style. Note that unlike common properties and styles, you cannot use the DISPLAY statement to invoke an ActiveX method specified in the Screen Section. You must use the MODIFY verb. ActiveX methods can take any number of parameters or no parameters. They can also take optional parameters (i.e., parameters that can be omitted). You specify the parameters in COBOL by enclosing them in parentheses. The optional parameters are always last. To invoke a method with no parameters, use empty parentheses ().

14. Each property or method name can be followed by '::' and then another property or method name to invoke methods in-line. "MethodName1::MethodName2" means invoke the method "MethodName1" of the current object and set the current object to the return value. When a property or method name is followed by a token other than '::', then it means to actually invoke the method on the current object passing the specified arguments or set the property to the specified value and reset the current object to null.

15. The MODIFY verb takes a control's home position (upper left corner), its handle, the name of an elementary Screen Section item, or '^', as its first parameter. Only the properties of the control that are specified in the MODIFY statement are updated.

16. The runtime automatically converts parameters to the appropriate types.

17. If a method has a return value, the runtime converts and moves it to the item specified in the GIVING clause.

18. You cannot use named parameters to avoid entering required parameters. You can omit optional parameters only.

19. You must specify only unnamed parameters before the BY NAME clause, and only named parameters after the BY NAME clause.

20. You can use one- and two-dimensional COBOL tables as property and method parameters for use in COM SAFEARRAYs. The runtime automatically converts the table to an COM SAFEARRAY, as long as it contains only one elementary item that is USAGE HANDLE or USAGE HANDLE OF VARIANT. See section 4.3.1 in *A Guide to Interoperating with ACUCOBOL-GT.*

21. Use the "AS *type-num"* phrase in the parameter expression if you want to force the parameter to be converted to a particular VARIANT type before it is passed to a property or method of an ActiveX control or COM object. You can tell from the object's documentation and the name of the parameter whether the object expects a particular VARIANT type, such as boolean.

    Use the AS phrase if the ActiveX or COM object requires a method or property parameter to be something different from the default VARIANT type chosen by the runtime for the particular COBOL data item or literal.

(See section 4.3 in *A Guide to Interoperating with ACUCOBOL-GT* for the rules that the runtime uses to determine the VARIANT type). Specify the word "AS" followed by a numeric literal or level 78 numeric constant that indicates the variant type to which you want the parameter converted. The "activex.def" COPY file in the ACUCOBOL-GT sample/def directory contains predefined level 78 constants for each of the VARIANT types.

Format 2 (MODIFY WINDOW)

1. A Format 2 MODIFY statement changes one or more attributes of an existing FLOATING or INITIAL WINDOW (not a subwindow). Attributes that are not specifically changed remain unchanged, except when a window is made larger, in which case it may also be repositioned in order to keep it on the screen. *Window-handle* or *generic-handle* identify the window to modify. If the WINDOW phrase is used and *generic-handle* is omitted, the current window is modified.

2. The LINE and COLUMN phrases specify the location of the window on the screen. The coordinates are relative to the interior space of the parent window. If the window being modified is the initial window, the coordinates are relative to its own interior. If either phrase is omitted, the corresponding row or column position is unchanged.

3. The AT phrase specifies both the row and column position. The first two or three digits of *screen-loc*, depending on the size of *screen-loc*, specify the row position. The remaining digits specify the column position. The values are treated in the same manner as in the LINE and COLUMN phrases. If either half of *screen-loc* is zero, the corresponding coordinate remains unchanged.

4. The SCREEN LINE and SCREEN COLUMN phrases set the location of the window. The coordinates indicate the absolute position desired on the screen. *Screen-line* and *screen-col* are given in the screen's base units. Base units are machine dependent. For character systems, the base unit is a character cell. For graphical systems, the base unit is a pixel. The upper left corner of the screen is position "1,1". The SCREEN LINE and SCREEN COLUMN phrases cannot be used if the LINE, COLUMN, or AT phrases are used.

5. The SIZE and LINES phrases change the size of the window. The dimensions indicate the interior of the window. The requested size must fit on the screen. If it does not, the size is not changed. After resizing the window, the runtime ensures that the window is fully visible on the screen. Resizing a window that has the RESIZABLE property will not change the window's physical dimensions if that window is not maximized. Note that only the window's logical dimensions are changed (thus increasing the scrolling region). The user will see the new size only if he or she later maximizes the window.

6. The TITLE phrase specifies a new title for the window. For this phrase to have an effect, the window must have a title area.

7. *Statement-1* executes if any part of the operation fails. An exception may be caused by one of the following situations:

   - The specified window size does not fit the screen. Note that this error occurs only on a non-Windows host. Because Windows allows you to have a desktop that is larger than the physical screen, you do not get an exception in this instance on Windows. You should use ACCEPT FROM TERMINAL-INFO to determine the maximum physical window size on a Windows host.

   - The window cannot be created, either because of an out-of-memory situation or the operating system fails to create it.

   - A window that has no input is activated.

   - An external window error occurs. For example, the window does not exist or cannot be created for some reason.

   - An illegal instruction is used.

8. *Statement-2* executes if the MODIFY statement succeeds.

9. The LAYOUT-MANAGER option attaches *manager* to the window.

10. The VISIBLE option makes a window visible or invisible. If the TRUE phrase is used, or *visible-state* is non-zero, then the window is made visible. Otherwise, it is made invisible.

11. The POP-UP MENU option changes the pop-up menu for the window. If *menu-1* is specified, then the corresponding menu becomes the new pop-up menu. If NULL is specified, any existing pop-up menu is removed (but not destroyed).

12. The Format 2 ENABLED phrase can be used to disable or enable a window. A user cannot interact with a disabled window.

13. The Format 2 EVENT PROCEDURE phrase changes the window's event procedure to *proc-1* (through *proc-2*, if specified). If the NULL option is used, then the window's event procedure, if any, is removed from the window. Additional information can be found in the **DISPLAY Statement** above and in **section 5.9.6**.

14. The ACTION phrase allows you to programmatically maximize, minimize, or restore a window. To use ACTION, assign it one of the following values (these names are found in acugui.def):

ACTION-MAXIMIZE	maximizes the window. It has the same effect as if the user clicked the "maximize" button. Allowed only for windows that have RESIZABLE or AUTO-RESIZE specified or implied for them.
ACTION-MINIMIZE	minimizes the window. Allowed only with INDEPENDENT windows that have the AUTO-MINIMIZE property set to true. It is not supported with other types of floating windows; if set, it is ignored by the runtime. ACTION-MINIMIZE has the same effect as if the user clicked the "minimize" button.
ACTION-RESTORE	If the window is currently maximized or minimized, restores the window to its previous size and position; otherwise, it has no effect. Allowed only for windows that can be maximized or minimized.

If you assign an ACTION value that is not allowed, then there is no effect other than to trigger the ON EXCEPTION phrase of the MODIFY statement (if present). Note that you can use the ACTION phrase to create a window that is initially maximized or minimized.

MOVE Statement

The MOVE statement transfers data to data items.

General Format

Format 1

```
MOVE source-item TO {dest-item} ...
```

Format 2

```
MOVE {CORRESPONDING} source-group TO dest-group
     {CORR         }
```

Format 3

```
MOVE alpha-item TO dest-item WITH {CONVERSION}
                                  {CONVERT   }

    [ ON EXCEPTION statement-1 ]

    [ NOT ON EXCEPTION statement-2 ]

    [ END-MOVE ]
```

Syntax Rules

1. *Source-item* is a literal or data item that represents the sending area.
2. *Dest-item* is a data item that receives the data.
3. *Source-group* and *dest-group* must be group items.
4. *Alpha-item* is a literal or data item of class alphanumeric.
5. CORR is an abbreviation for CORRESPONDING.
6. If *dest-item* is numeric or numeric edited, *source-item* may not be HIGH-VALUES, LOW-VALUES, SPACES, or QUOTES.
7. *Source-item* and *dest-item* must be of a compatible category. See General Rule 9 below.
8. *Statement-1* and *statement-2* are imperative statements.

General Rules

1. A Format 1 MOVE statement moves *source-item* to each *dest-item* in the same order in which they appear in the statement.

2. Subscript evaluation for *source-item* occurs once before the move to the first *dest-item*.

3. Subscript evaluation for *dest-item* occurs immediately before the move to that item.

4. The length of *source-item* is computed once immediately before the move to the first *dest-item*. The compiler option "-Dz" modifies size checking rules for numeric moves.

5. The length of *dest-item* is computed immediately before the MOVE to that item.

6. Reference modification is allowed on *source-item* and *dest-item*.

7. *Source-item* and *dest-item* should not overlap (reference the same location in memory). The compiler does not detect this condition. The results of such a move are undefined. One possible outcome is a memory access violation. For example:

   ```
   MOVE myStr(2:myLen - 1) to myStr(1:myLen)
   ```

   attempts to move part of *myStr* to *myStr*, which gives undefined results. Another mistake is to create an overlap by moving the value of an item to a REDEFINES of the same item.

8. When the CORRESPONDING phrase is used, selected elementary items in *source-group* are moved to corresponding items in *dest-group*. This is treated as a series of Format 1 MOVE statements, one for each corresponding pair of data items.

9. The effects and legality of a MOVE statement depend on the category of the *source-item* and *dest-item*. Data items are assigned a category according to their PICTURE clause. Literals are assigned a category based on the following rules:

   a. Numeric literals are numeric. Nonnumeric literals are alphanumeric.

b. The figurative constant ZERO is numeric when *dest-item* is numeric or numeric edited, otherwise it is alphanumeric.

c. The figurative constant SPACE is alphabetic.

d. All other figurative constants are alphanumeric.

10. Any Format 1 MOVE statement that has a group item as either a source or destination item is treated as a simple alphanumeric to alphanumeric move. (No implied conversion is implied.) Any category of data may be the source or destination of a group item MOVE.

11. The following table outlines the combinations of *source-item* and *dest-item* that are allowed by the MOVE statement. The numbers in the table are the "General Rules" numbers in this section where each combination is described:

Sending Category:	**Receiving Item Category:**		
	Alphabetic	**Alphanumeric/ Alphanumeric Edited**	**Numeric / Numeric Edited**
Alphabetic	Yes (12)	Yes (13)	No (15)
Alphanumeric	Yes (12)	Yes (13)	Yes (14)
Alphanumeric Edited	Yes (12)	Yes (13)	No (15)
Numeric Integer	No (15)	Yes (13)	Yes (14)
Numeric Non-integer	No (15)	No (15)	Yes (14)
Numeric Edited	No (15)	Yes (13)	Yes (14)

12. When *dest-item* is alphabetic, justification and space filling occur according to the standard alignment rules.

13. When *dest-item* is alphanumeric or alphanumeric edited, justification and space filling occur according to the standard alignment rules. If *source-item* is signed numeric, the operational sign is not moved. If the sign occupies a separate character position, that sign character is not moved, and the size of *source-item* is treated as being one less.

14. When *dest-item* is numeric or numeric edited, decimal point alignment and zero filling occur according to the standard alignment rules. If *source-item* is unsigned, it is treated as being positive. If *dest-item* is unsigned, the absolute value of *source-item* is moved. If *dest-item* is signed, its sign is set to the sign of *source-item*. If *source-item* is numeric edited, it is "de-edited" first such that *dest-item* receives the same numeric value.

15. The following moves are illegal:

    a. An alphabetic or alphanumeric edited data item may not be moved to a numeric or numeric edited data item.

    b. A numeric or numeric edited data item may not be moved to an alphabetic item.

    c. A non-integer numeric data item cannot be moved to an alphanumeric or alphanumeric edited data item.

16. A Format 3 MOVE statement performs a logical conversion of *alpha-item* into the format of *dest-item*. *Dest-item* may be any type of data item. This is normally done to convert a character representation of a number into the corresponding numeric value. The rules of conversion are the same as the rules used by the CONVERT option of the ACCEPT statement. For a detailed description of these rules, see **section 6.4.9, "Common Screen Options,"** under the subheading "CONVERT phrase."

17. If the ON EXCEPTION phrase is specified, then *statement-1* executes when a conversion error occurs. If a conversion error occurs, then the value assigned to *dest-item* is the value determined by ignoring the illegal characters in *alpha-item*. If the NOT ON EXCEPTION phrase is specified, then *statement-2* executes when no conversion error occurs.

MULTIPLY Statement

The MULTIPLY statement performs arithmetic multiplication.

General Format

Format 1

```
MULTIPLY source BY { result [ROUNDED] } ...

    [ ON SIZE ERROR statement ]

    [ NOT ON SIZE ERROR statement ]

    [ END-MULTIPLY ]
```

Format 2

```
MULTIPLY source BY source

      GIVING { result [ROUNDED] } ...

    [ ON SIZE ERROR statement ]

    [ NOT ON SIZE ERROR statement ]

    [ END-MULTIPLY ]
```

Syntax Rules

1. *Source* is a numeric literal or numeric data item.
2. *Result* is a numeric or numeric edited data item. In Format 1, *result* may not be numeric edited.
3. *Statement* is an imperative statement.

General Rules

1. In Format 1, each *result* is multiplied by *source*. The product is stored back in *result*.
2. In Format 2, the two *source* operands are multiplied together. The product is stored in each *result* variable.
3. Additional information can be found in the sections covering Arithmetic Operations (**section 6.4.1**), Multiple Receiving Fields (**section 6.4.2**), the ROUNDED option (**section 6.4.3**), and the SIZE ERROR option (**section 6.4.4**).

NEXT SENTENCE Statement

The NEXT SENTENCE statement causes control to be transferred to the next COBOL sentence (following the next period). This is distinct from the logically next COBOL verb, which is the result of a CONTINUE statement. Note that the ANSI standard states that "NEXT SENTENCE is an archaic feature and its use should be avoided."

General Format

```
NEXT SENTENCE
```

Syntax Rules

A NEXT SENTENCE statement is allowed anywhere a conditional statement or imperative-statement is allowed.

General Rules

A NEXT SENTENCE statement transfers the flow of execution to the logically next COBOL verb following the next period.

OPEN Statement

The OPEN statement initiates processing of a file.

General Format

```
OPEN [EXCLUSIVE]

    { {INPUT  } { file [WITH NO REWIND] } ... } ...
      {OUTPUT}        [lock-option   ]
      {I-O    }
      {EXTEND}
```

lock-option may be either of these formats:

```
    { ALLOWING {NO OTHERS} }
               {READERS  }
               {WRITERS  }
               {UPDATERS }
```

```
                {ALL       }

{ {WITH}    {LOCK          }    }
  {FOR }    {MASS-UPDATE   }
            {BULK-ADDITION }
```

Syntax Rules

1. *File* is the name of a file described in the Data Division. It may not be a sort file.
2. The I-O phrase can be specified only for files that reside on disk.
3. The MASS-UPDATE and BULK-ADDITION phrases may be specified only for indexed files and may not be specified along with the INPUT phrase.
4. The WITH NO REWIND phrase may be specified only for sequential files opened with the INPUT or OUTPUT phrase.
5. If EXCLUSIVE is specified, then *lock-option* may not be specified.

General Rules

1. A successful OPEN statement prepares the file for further processing by other I/O statements and puts the file in its open mode.
2. An OPEN may not be performed on a file that is already open. A file is not open when the program is in its initial state or when the last I/O statement on the file was a successful CLOSE statement.
3. Except for the OUTPUT option, and as noted in rule 19 below, a file's organization and data description must match those used when the file was created.
4. The following table indicates the effects of the various OPEN types on files that are and are not available. A file is not available if it does not exist on the host computer. Note that different results occur for missing files if the file's SELECT contains the OPTIONAL phrase.

Mode	**Available**	**Unavailable**
INPUT	Open succeeds	Open fails

Mode	Available	Unavailable
- Optional	Open succeeds	Open succeeds
I-O	Open succeeds	Open fails
- Optional	Open succeeds	Open creates file
OUTPUT	Open recreates file	Open creates file
EXTEND	Open succeeds	Open fails
- Optional	Open succeeds	Open creates file

5. The different OPEN types allow for different I/O statements to be used. This is summarized in the following chart.

Statement	Input	Output	I-O	Extend
READ	X		X	
WRITE		X	X	X
REWRITE			X	
START	X		X	
DELETE			X	
UNLOCK	X		X	

6. After a successful OPEN statement, the file is positioned at its first logical record. (The only exception to this rule is that a sequential access file opened with the EXTEND phrase is positioned after its last record. Records written to such a file are appended to the file.)

7. A file opened with the OUTPUT phrase is logically recreated. This is equivalent to physically removing the file (if it exists) and creating it anew.

8. If a file is not available, the OPTIONAL phrase is specified in the file's SELECT and the file is opened with the INPUT phrase, then the first READ on the file will return an end-of-file status condition.

9. The execution of the OPEN statement causes the I-O status associated with *file* to be updated. Note that opening an OPTIONAL file when that file does not exist will return a different status code than opening it if it does exist.

10. An OPEN statement with multiple *files* is equivalent to a series of OPEN statements referencing the files in the order named.

11. The open state of a file is not affected by other instances of the same file in other run units or other programs of the same run unit. These other files may be opened or closed without affecting the ability of the current program to open the same file, except as modified by the file locking rules below.

12. The EXCLUSIVE, WITH, and ALLOWING phrases allow for various forms of file locking. There are three forms of file locking: ALLOWING ALL, ALLOWING READERS, and ALLOWING NO OTHERS. File locking is enforced for all file types residing on disk but may not be enforced for non-disk files for some operating systems.

Note: The "-Fn" compiler option specifies file locking as the default behavior for files that do not have locking or sharing already specified or implied from within the program. See the Users' Guide, **section 2.2.7, "File Options"** for details on this and other options related to file locking.

---

13. The following phrases imply the ALLOWING ALL form of file locking:

    a. No *lock-option* specified

    b. ALLOWING ALL

    c. ALLOWING WRITERS

    d. ALLOWING UPDATERS

    This file locking mode indicates that other programs can access the file without restriction except that another program may not execute an OPEN OUTPUT while this program keeps the file open.

14. The following phrases imply the ALLOWING READERS form of file locking:

    a. ALLOWING READERS

    b. EXCLUSIVE or WITH LOCK specified with the INPUT phrase

A file open in this mode does not allow any other program to open this file other than with the INPUT phrase. Furthermore, this OPEN will fail if any other programs currently have the file open unless the INPUT phrase was used by all of these other programs.

15. These phrases imply the ALLOWING NO OTHERS form of file locking:

    a. ALLOWING NO OTHERS

    b. EXCLUSIVE or WITH LOCK specified with the OUTPUT, I-O, or EXTEND phrases

    c. WITH MASS-UPDATE

    d. BULK-ADDITION

    This form of file locking does not allow any other programs to open the file, and this OPEN will fail if any other programs currently have the file open.

16. The WITH MASS-UPDATE phrase is equivalent to the WITH LOCK phrase with some additional effects. It may be specified only for indexed files. This phrase indicates to the runtime system that the file in question will be heavily updated by this program. The runtime system may be able to use this information to access the file more efficiently. Book 1, *ACUCOBOL-GT User's Guide*, section 6.1.6, "Indexed File Considerations," contains more information about the effects of this phrase and advice on when it should be specified.

17. The BULK-ADDITION phrase is equivalent to the MASS-UPDATE phrase with some additional effects. For Vision files, the BULK-ADDITION phrase opens the file in "bulk addition" mode, which substantially increases efficiency when you are writing a large number of new records to the file. BULK-ADDITION has several significant effects, including some changes to standard COBOL file handling rules. See section 6.1.6.3 of *ACUCOBOL-GT User's Guide* for details. For host file systems other than Vision, specifying BULK-ADDITION has the same effect as specifying MASS-UPDATE. In this case, none of the special handling dictated by BULK-ADDITION applies.

18. A file with a LOCK MODE IS EXCLUSIVE phrase in its SELECT treats all OPEN statements as if they were written with the EXCLUSIVE option.

19. The configuration variable EXTRA_KEYS_OK allows you to open an indexed file without specifying all of that file's alternate keys. For more information, see the listing for **EXTRA_KEYS_OK** in Appendix H, Book 4, *Appendices*.

20. Note that the IBM DOS/VS COBOL "-Cv" compatibility mode supports Reversed File Reads.

PERFORM Statement

The PERFORM statement executes a procedure with optional loop control.

General Format

Format 1

```
PERFORM [ IN THREAD ]

      [ procedure-1 [ {THROUGH} procedure-2 ] ]
                      {THRU   }

      [ HANDLE IN handle-1 ]

      [ statement END-PERFORM ]
```

Format 2

```
PERFORM [ IN THREAD ]

      [ procedure-1 [ {THROUGH} procedure-2 ] ]
                      {THRU   }

      [ HANDLE IN handle-1 ]

        number TIMES

      [ statement END-PERFORM ]
```

Format 3

```
PERFORM [ IN THREAD ]

      [ procedure-1 [ {THROUGH} procedure-2 ] ]
                      {THRU   }

      [ HANDLE IN handle-1 ]

      [ WITH TEST {BEFORE} ] UNTIL condition
                  {AFTER }

      [ statement END-PERFORM ]
```

Format 4

```
PERFORM [ IN THREAD ]

      [ procedure-1 [ {THROUGH} procedure-2 ] ]
                      {THRU   }

      [ HANDLE IN handle-1 ]

      [ WITH TEST {BEFORE} ]
                  {AFTER }

        VARYING counter FROM starting-val

            BY increment UNTIL condition

      [ AFTER counter FROM starting-val

            BY increment UNTIL condition ] ...

      [ statement END-PERFORM ]
```

Syntax Rules

1. *Handle-1* is a HANDLE or HANDLE OF THREAD data item.
2. *Procedure-1* and *procedure-2* are paragraph or section names in the Procedure Division.
3. *Statement* is an imperative statement.

4. *Number* is an integer numeric literal or data item. The value of *number* cannot exceed 2,147,483,647.
5. *Condition* is a conditional expression.
6. *Counter* is a numeric data item.
7. *Starting-val* is a numeric literal or data item.
8. *Increment* is a non-zero numeric literal or data item.
9. A PERFORM statement must have exactly one of the *procedure-1* or the *statement* END-PERFORM phrases specified or a period (".") for an implied END-PERFORM.
10. The words THRU and THROUGH are interchangeable.

General Rules

1. A PERFORM statement that contains *procedure-1* is an out-of-line PERFORM. When *statement* is used instead, then it is an in-line PERFORM.
2. An in-line PERFORM statement functions according to the same rules for an otherwise identical out-of-line PERFORM except that *statement* is executed in place of the statements in the range of *procedure-1* (through *procedure-2* if specified).
3. When the PERFORM executes, control transfers to the first statement of *procedure-1*. Control might not transfer, however, depending on the evaluation of *condition* (if specified). The PERFORM statement also establishes an implicit transfer of control to the end of the PERFORM statement according to the following rules:
    a. If *procedure-1* is a paragraph name, and *procedure-2* is not specified, the return is after the last statement of *procedure-1*.
    b. If *procedure-1* is a section name, and *procedure-2* is not specified, the return is after the last statement of the last paragraph of *procedure-1*.
    c. If *procedure-2* is specified and is a paragraph name, the return is placed after the last statement of *procedure-2*.

d. If *procedure-2* is specified and is a section name, the return is placed after the last statement of the last paragraph of *procedure-2*.

e. If an in-line PERFORM is specified, an execution of the PERFORM statement is completed after *statement* has executed.

4. *Procedure-1* and *procedure-2* are not necessarily related except that control starts at *procedure-1* and returns when it reaches the end of *procedure-2*. In particular, GO TO and PERFORM statements may occur between *procedure-1* and the end of *procedure-2*.

5. Control can pass to statements that are inside the range of *procedure-1* through *procedure-2* by mechanisms other than PERFORM. In this case, the implicit return to the PERFORM referencing these statements is not made. An implicit return occurs only for an active PERFORM.

6. The range of a PERFORM statement consists of those statements that are executed as a result of executing that PERFORM. This includes statements that are executed as the result of GO TO and PERFORM statements included in the range of the PERFORM statement.

7. If that range of a PERFORM statement includes another PERFORM statement, the range of the included PERFORM must be either totally included in or totally excluded from the logical sequence of the first PERFORM statement. Thus an active PERFORM included in the range of another active PERFORM may not allow control to pass to the return point of the first PERFORM. Furthermore, two or more active PERFORM statements may not have a common return point.

8. Within a thread, a paragraph under the control of a PERFORM statement may (directly or indirectly) PERFORM itself only if the compile-time option "-Zr1" is specified (this option is specified by default).

9. If the TEST phrase is not specified, TEST BEFORE is implied.

10. When the THREAD option is used, a new thread is created by the PERFORM statement. Once control returns to the end of the PERFORM statement, the thread is terminated. Note that all of the statements contained in the scope of the PERFORM are executed in the new thread. This includes any loop control operations implied by the PERFORM. For example, the statement:

```
PERFORM THREAD, PARA-1 5 TIMES
```

creates a single thread the performs PARA-1 five times (as opposed to creating five separate threads, each of which executes PARA-1 once).

11. If *handle-1* is specified, the new thread's unique ID is stored in *handle-1*.

Format 1

A Format 1 PERFORM statement executes its range exactly once.

Format 2

A Format 2 PERFORM statement executes its range a fixed number of times. If *number* is zero or negative, control passes to the end of the PERFORM statement. Otherwise the range of the PERFORM statement executes *number* times. Changing the value of *number* during the execution of the PERFORM statement does not change the number of times that range is executed.

Format 3

A Format 3 PERFORM statement executes its range until *condition* evaluates "true". If TEST BEFORE is specified or implied, the evaluation of *condition* occurs prior to any executions of the PERFORM range. Thus if *condition* is true when the PERFORM starts, the range will not be executed. If TEST AFTER is specified, the evaluation of *condition* does not occur until after the first execution of the PERFORM range.

Format 4

1. A Format 4 PERFORM statement executes its range a variable number of times while systematically changing the value of one or more variables.
2. If TEST BEFORE is specified or implied and only one *counter* is specified:
    a. *Counter* is set to the value of *starting-val* when the PERFORM statement begins.

b. If *condition* is false, the PERFORM range executes once. Then *increment* is added to *counter* and *condition* is evaluated again. This cycle repeats until *condition* is true.

c. If *condition* is true when the PERFORM statement begins executing, control is passed to the end of the statement after *counter* is set to *starting-val*.

3. If TEST BEFORE is specified or implied and two or more *counters* are used:

   a. Each *counter* is set to the value of the corresponding *starting-val*.

   b. If the first *condition* is true, control transfers to the end of the PERFORM statement.

   c. If the last *condition* is false, the range of the PERFORM executes once. The final *counter* is incremented by the corresponding *increment* and the last *condition* is evaluated again. This cycle continues until the last *condition* is true.

   d. When the last *condition* is true, the last *counter* is set again to the corresponding *starting-val*. The preceding *counter* is then incremented by the corresponding *increment* and the preceding *condition* is evaluated. If the *condition* is false, step (c) is performed again.

   e. When the *condition* in step (d) is true, the cycle repeats for the next higher-level *counter*. These cycles continue repeating in this hierarchical manner until the topmost (VARYING) *counter* is cycled. For each level, all levels underneath it are reinitialized and run through a full cycle each time the corresponding *counter* is incremented.

   f. The PERFORM statement ends when the uppermost (the first) *condition* evaluates true.

4. At the end of a PERFORM with the TEST BEFORE phrase, the value of the first *counter* exceeds the last-used value by one addition of *increment*. The values of all other *counters* are equal to their corresponding *starting-val*.

5. If the TEST AFTER phrase is specified and only one *counter* is used:

   a. *Counter* is set to the value of *starting-val*.

b. The range executes once. Then *condition* is evaluated. If it is false, *increment* is added to *counter* and the range executes again. This cycle continues until *condition* is true.

6. If the TEST AFTER phrase is specified and two or more *counters* are used:

   a. Each *counter* is set to its corresponding *starting-val*.

   b. The PERFORM range executes once. The last *condition* is then evaluated. If it is false, the last *counter* is incremented by its corresponding *increment* and the PERFORM range executes again. This continues until the last *condition* evaluates true.

   c. When the last *condition* is true, the preceding *condition* is evaluated. If it is false, the value of the corresponding *counter* is incremented by its *increment*, the last *counter* is set to its corresponding *starting-val*, and step (b) is performed through another cycle.

   d. When the *condition* in step (c) is true, the cycle repeats for the next higher-level *counter*. These cycles continue repeating in this hierarchical manner until the topmost (VARYING) *counter* is cycled. For each level, all levels underneath it are reinitialized and run through a full cycle each time the corresponding *counter* is incremented.

   e. The PERFORM statement ends when the topmost (VARYING) *condition* is true.

7. At the end of a PERFORM statement with the TEST AFTER phrase, the value of each *counter* is the same as at the end of the most recent execution of the PERFORM range.

READ Statement

The READ statement makes records available to the program from the program's data files.

General Format

Format 1

```
READ file-name  [NEXT    ] RECORD
                [PREVIOUS]
                [BACKWARD]

          [ WITH [NO  ] LOCK ]
                 [KEPT]

          [ INTO dest ]

          [ ALLOWING UPDATERS ]

          [ AT END statement-1 ]

          [ NOT AT END statement-2 ]

          [ END-READ ]
```

Format 2

```
READ file-name RECORD

     [ WITH [NO  ] LOCK ]
            [KEPT]

     [ INTO dest ]

     [ ALLOWING UPDATERS ]

     [ KEY IS key-name ]

     [ INVALID KEY statement-1 ]

     [ NOT INVALID KEY statement-2 ]

     [ END-READ ]
```

Syntax Rules

1. *File-name* is the name of a file described in the Data Division. It may not be a sort file.

2. *Dest* is a data item.
3. *Key-name* is the name of a data item specified as a record key for *file-name*.
4. *Statement-1* and *statement-2* are imperative statements.
5. Format 1 must be used for sequential access files.
6. The KEY phrase can be used only for indexed files.
7. *Dest* may not occupy any of the storage area used by the record area of *file-name*.
8. BACKWARD and PREVIOUS are equivalent.
9. The NEXT or PREVIOUS phrase must be specified for a Format 1 READ for dynamic access mode files.
10. The PREVIOUS phrase may not be specified for a sequential organization file.
11. The word KEPT is treated as commentary.
12. The LOCK, INTO and ALLOWING phrases may appear in any order.

General Rules

1. The file referenced by a READ statement must be open in the INPUT or I-O mode when the statement executes.
2. For sequential access mode files, if neither NEXT nor PREVIOUS is used, NEXT is implied.
3. A successful READ statement causes the file's record area to be filled with the record retrieved from the file.
4. If the record read is smaller than the record area, the excess characters are left unmodified unless the file has automatic trailing space removal specified. In this case, the record is padded with spaces.
5. The READ statement updates the value of the associated FILE STATUS variable.
6. A successful Format 1 READ statement retrieves a record from the file according to the following rules:

a. The last OPEN, READ, or START verb used for the file determines which record is retrieved. Other file operations do not affect which record is retrieved.

b. If an OPEN verb was the last verb to affect the file position, then the first record is retrieved if the NEXT phrase is used (or implied). If the PREVIOUS phrase is used, an end-of-file condition occurs.

c. If the last verb to affect the file position was a successful START statement, then the record selected by that START statement is returned, regardless of whether the NEXT or PREVIOUS phrase was used.

d. If the last verb to affect the file position was a successful READ statement, then the following or the preceding record is retrieved, depending on the NEXT or PREVIOUS phrase used.

e. For sequential and relative files, the record ordering is based on the physical ordering of the records in the file (relative files are physically ordered by ascending record numbers).

f. For indexed files, the record ordering is based on the logical ordering of the current Key of Reference. The Key of Reference is set by the last successful OPEN, READ, or START statement executed for the file.

7. When a Format 1 READ statement executes, the preceding rule may indicate that no next logical record exists. When this happens, the following occurs:

   a. The at-end condition is set and the appropriate FILE STATUS is set.

   b. If the AT END phrase is specified, *statement-1* executes. Control does not proceed to a USE AFTER EXCEPTION statement.

   c. If no AT END phrase is specified, but an appropriate USE AFTER EXCEPTION procedure exists, that procedure is executed with an implied return to the end of the READ statement.

   d. If neither case (b) nor (c) applies, then an error message is printed and the program halts.

8. If the at-end condition does not occur, and no other exception causes the USE AFTER EXCEPTION procedure to execute, the NOT AT END phrase (if any) is used and *statement-2* is executed.

9. For a relative file, a Format 1 READ updates the contents of the file's RELATIVE KEY data item to reflect the record number of the returned record.

10. When the program is sequentially accessing records from an indexed file that contains records with duplicated alternate key values, those records are returned in the same order in which they were created. These duplicate values can be created by WRITE or REWRITE statements. (These records may be reordered in the process of rebuilding the file on another key.)

    For sites using the RMS file system, please note that when a set of records having duplicate keys is encountered, RMS returns only the first record in the set.

11. A Format 2 READ statement provides you with the ability to read records in random order by specifying appropriate key values. A Format 2 READ statement on a relative file retrieves the record whose record number is specified by the file's RELATIVE KEY data item.

12. For indexed files, a Format 2 READ statement retrieves the record that contains the same key value as the corresponding data item in the file's record area. The key used is the one named in the KEY phrase of the READ statement. If no KEY phrase is used, the file's primary key is implied. The key used becomes the file's current Key of Reference for future Format 1 READ statements. For key values that are duplicated, the record that corresponds to the first of the sequence of duplicated values (as described in General Rule 10 above) is returned.

13. After successfully retrieving a record, a Format 2 READ statement sets the file's File Position Indicator to the next logical record according to General Rule 6.

14. If a Format 2 READ cannot find a record with the appropriate key value, the invalid-key condition exists. When this happens the following occurs:

a. If the INVALID KEY phrase is specified, *statement-1* executes. Control does not proceed to a USE AFTER EXCEPTION statement.

b. If no INVALID KEY phrase is specified, but an appropriate USE AFTER EXCEPTION procedure exists, that procedure is executed with an implied return to the end of the READ statement.

c. If neither case (a) nor (b) applies, then an error message is printed and the program halts.

15. If the NOT INVALID KEY phrase is used and the invalid-key condition does not exist, and no other condition causes a USE AFTER EXCEPTION procedure to execute, *statement-2* is executed.

16. If an applicable USE AFTER EXCEPTION procedure exists, it executes whenever a condition occurs that results in a non-zero file status. However, it does not execute if the condition is invalid-key and an INVALID KEY phrase is used, or if the condition is at-end and an AT END phrase is used.

17. If a READ statement is unsuccessful, the current file position and the current Key of Reference are both set to be undefined. See General Rules 22 and 23 for exceptions.

18. The INTO phrase causes the contents of the file's record area to be moved to *dest* according to the rules of the MOVE statement. This move occurs after the record is retrieved, but only if the statement is successful.

19. The WITH NO LOCK and ALLOWING UPDATERS phrases are equivalent. They both cause the record to be read without record locking. In the default mode, any successful READ on a file open in the I-O mode causes the retrieved record to be locked. A locked record may not be read (with lock) or updated by another program. Once locked, a record remains locked until any other I/O statement in the program that locked it is executed for the file. (An exception to this is files that hold multiple record locks--see **section 4.3.1, "File-Control Paragraph,"** for details.) Once another I/O statement is executed for the file, the currently locked record becomes unlocked, even if the I/O is unsuccessful.

20. The WITH NO LOCK and ALLOWING UPDATERS phrases are implied for a file open in the INPUT mode, and thus have no effect.

21. For files with manual record locking mode (see **section 4.3.1, "File-Control Paragraph,"**) the WITH NO LOCK phrase is implied. For such a file to place a record lock, it must specify WITH LOCK on the READ statement.

22. Normally, a read that fails due to a record lock will return the appropriate FILE STATUS. In RM/COBOL compatibility mode, however, if the file has no applicable USE AFTER EXCEPTION procedure available, the program will wait until the record becomes unlocked. It will then read the record and proceed normally. Note that this can result in deadlock. This feature is provided for RM/COBOL compatibility. Because of the danger in using it, it is not recommended.

23. The current Key of Reference and current file position are not modified by a record locked condition. This allows a program to wait an appropriate amount of time for the record to become unlocked and then try executing the same READ statement without having to re-establish the current file position. Because of the nature of RMS, this rule is not followed for a program running under the VMS operating system. In this case, the File Position Indicator is undefined.

24. If the end of a file is reached by a READ NEXT statement, a subsequent READ PREVIOUS statement will return the last record in the file. Similarly, if the beginning of a file is reached by a READ PREVIOUS statement, a READ NEXT statement will retrieve the first record of the file.

25. If the NEXT SENTENCE option is used, control passes to the next executable sentence. Note that the ANSI standard states that "NEXT SENTENCE is an archaic feature and its use should be avoided."

26. The IBM DOS/VS COBOL "-Cv" compatibility mode supports Reversed File Reads.

RECEIVE Statement

The RECEIVE statement retrieves messages sent by other threads.

General Format

```
RECEIVE dest-item FROM { THREAD thread-1 }
```

```
                                { LAST THREAD     }
                                { ANY THREAD      }
```

Remaining phrases are optional, can appear in any order.

```
{ BEFORE TIME timeout }
{ WITH NO WAIT        }

THREAD IN thread-2

SIZE   IN size-item

STATUS IN status-item

[ ON EXCEPTION statement-1 ]

[ NOT ON EXCEPTION statement-2 ]

[ END-RECEIVE ]
```

Syntax Rules

1. *Dest-item* is any data item.
2. *Thread-1* and *thread-2* are usage HANDLE or HANDLE OF THREAD data items. *Thread-2* may not be indexed or reference modified.
3. *Timeout* is a numeric literal or data item.
4. *Size-item* is a numeric data item. It may not be indexed or reference modified.
5. *Status-item* is a two-character group item, PIC XX, or PIC 99 data item. It may not be indexed or reference modified.
6. *Statement-1* and *statement-2* are any imperative statements.

General Rules

1. The RECEIVE statement returns the next available message into *dest-item*. Only messages from the proper source are allowed as follows:

a. FROM THREAD *thread-1* specifies that only messages from the thread identified by *thread-1* are allowed.

b. FROM LAST THREAD specifies that only messages from the *last thread* are allowed (see section 6.8.1, Book 1, *ACUCOBOL-GT User's Guide* for a discussion of the *last thread*).

c. FROM ANY THREAD specifies that all messages are allowed.

2. Messages are received in the order sent. If a message is available when the RECEIVE statement executes, the RECEIVE statement finishes immediately. Otherwise, the RECEIVE statement waits for a message to become available. This provides an efficient method for threads to synchronize with each other.

3. When BEFORE TIME is specified, the RECEIVE statement will time out after the specified (*timeout*) number of hundredths of seconds. If the RECEIVE statement times out before receiving a message, it terminates with an exception condition and does not modify *dest-item*, *thread-2*, or *size-item*. If *timeout* is zero, then the RECEIVE statement times out immediately if a message is not available. Specifying NO WAIT is equivalent to specifying a *timeout* value of zero.

4. If no message is available and the sending thread (as specified by rule1 above) does not exist or terminates before sending a message, the RECEIVE statement terminates with an exception condition and does not modify *dest-item* or *size-item*. This condition is reflected in the status placed in *status-item*. Note that the test occurs before the time-out test in the case that *timeout* is zero or NO WAIT is specified.

5. The RECEIVE statement places the thread ID of the sending thread in *thread-2*.

6. The size of the message sent is placed in *size-item*. The size is expressed in standard character positions (bytes). If the message is longer than *dest-item*, it is truncated. If the message is shorter than *dest-item*, it is padded on the right with spaces.

7. The status of the RECEIVE statement is placed in *status-item*. The following values are possible (these approximate the standard file status codes):

   "00" Success - message received

   "04" Success - message received, but it was truncated

   "10" Exception - sending thread does not exist or terminated

   "99" Exception - timed out

8. If the RECEIVE statement is successful (as indicated in rule 7 above), *statement-2* executes. If an exception condition is returned, *statement-1* executes.

RELEASE Statement

The RELEASE statement makes records available to the initial phase of a sort operation.

General Format

```
RELEASE record  [ FROM source-rec ]
```

Syntax Rules

1. *Record* is the name of a record of a sort file described in an SD entry in the File Section.
2. *Source-rec* is a data item.
3. A RELEASE statement can appear only in an input procedure to a SORT verb.
4. *Record* and *source-rec* may not have overlapping storage.

General Rules

1. The RELEASE statement makes *record* available to the input phase of a sort operation. For details, see the entry in this section for the "SORT Statement."

2. The RELEASE statement may be executed only while the program is executing an input procedure for a SORT verb that is sorting the file associated with *record*. Any other use causes a runtime error.

3. If the FROM phrase is used, *source-rec* is moved to *record* according to the rules of the MOVE statement before *record* is released to the sort operation.

4. The size of the record released to the sort operation is determined by the size of *record*.

5. If a RELEASE statement is executed in a wrong context, e.g., outside an input procedure, the runtime displays the error "Illegal RELEASE." This error belongs to the class of "intermediate" runtime errors that, upon occurrence, call installed error procedures. See Book 4, *Appendices*, Appendix I "Library Routines," CBL_ERROR_PROC for details.

RETURN Statement

The RETURN statement retrieves records from sort or merge operations.

General Format

```
RETURN file-name RECORD [ INTO dest-record ]

      AT END statement-1

    [ NOT AT END statement-2 ]

    [ END-RETURN ]
```

Syntax Rules

1. *File-name* is the name of a sort file described by an SD entry in the Data Division.

2. *Dest-record* is a data item.

3. *Statement-1* and *statement-2* are imperative statements.

4. A RETURN statement may appear only in an output procedure associated with a SORT or MERGE statement. A RETURN statement may not be placed in Declaratives.

5. The record area associated with *file-name* and *dest-record* may not have overlapping storage.

General Rules

1. The RETURN statement returns the next record from the output phase of a SORT or MERGE statement and places this record in the record area associated with *file-name*. The RETURN statement may be executed only while the program is executing an output procedure of a SORT or MERGE statement.

2. Any area in the record area associated with *file-name* that is beyond the end of the returned record is left unchanged.

3. If the INTO phrase is specified, the record area associated with *file-name* is moved to *dest-record* according to the rules of the MOVE statement. This move occurs after the record is retrieved, but only if the statement is successful.

4. When no more records are available from the SORT or MERGE operation, *statement-1* is executed. Otherwise, *statement-2*, if specified, is executed.

5. The RETURN statement does *not* update the FILE STATUS variable associated with *file-name*.

6. If a RETURN statement is executed in a wrong context, the runtime displays the error "Illegal RETURN." This error belongs to the class of "intermediate" runtime errors that, upon occurrence, call installed error procedures. See Book 4, *Appendices*, Appendix I "Library Routines," CBL_ERROR_PROC for details.

REWRITE Statement

The REWRITE statement logically replaces a record in a file.

General Format

```
REWRITE record [ FROM source-field ]

    [ INVALID KEY statement-1 ]

    [ NOT INVALID KEY statement-2 ]

    [ END-REWRITE ]
```

Syntax Rules

1. *Record* must be the name of a logical record in the Data Division File Section. The associated file may not be a sort file.
2. *Source-field* is a data item or literal.
3. *Statement-1* and *statement-2* are imperative statements.
4. The INVALID KEY and NOT INVALID KEY phrases may not be specified for sequential files or relative files with sequential access.
5. *Record* and *source-field* may not share any storage area.

General Rules

1. The file associated with *record* must be a mass storage file and must be open in the I-O mode.
2. For files with sequential access mode, the preceding I/O statement executed for the file must have been a successful READ statement. The REWRITE statement replaces the last record read by the contents of *record*. If the file is an indexed file, the primary key must not have been changed since the last READ.
3. For random or dynamic access mode files, the REWRITE statement replaces the record specified by the file's key.

   For relative files, this is the record specified by its RELATIVE KEY data item. For indexed files, the record identified by the primary key is replaced.
4. For an indexed file with alternate keys, the order in which duplicated keys are subsequently returned is affected as follows:

a. If the value of an alternate key has not changed, its order of retrieval is unchanged.

b. If the value is changed, and the new value is a duplicated value, the record's logical position is unpredictable within the set of records with that value.

5. The REWRITE statement does not affect the current file position.

6. The following occurrences cause the invalid-key condition:

   a. The access mode is sequential and an indexed file's primary key is not identical to the value returned from the preceding READ statement.

   b. The record being replaced does not exist in the file.

   c. The value of an alternate key that does not allow duplicates equals that of another record already in the file.

   The invalid-key condition causes the REWRITE to fail and does not update the file.

7. If the invalid-key condition occurs, and there is an INVALID KEY phrase, *statement-1* executes. If there is no INVALID KEY phrase, but there is an appropriate USE AFTER EXCEPTION procedure, that procedure executes. Otherwise, an invalid-key condition causes a message to be printed and the program halts.

8. If the NOT INVALID KEY phrase is specified, *statement-2* executes if the REWRITE statement is successful.

9. For a sequential file, the size of the record must be the same as the one it is replacing. The size of the record written is determined by the size of *record*.

10. The REWRITE statement updates the value of the FILE STATUS data item for the file.

11. If the FROM phrase is specified, it is identical to first moving the value of *source-field* to *record* using the rules of the MOVE statement and then performing the REWRITE as if there were no FROM phrase.

ROLLBACK Statement

The ROLLBACK TRANSACTION verb causes a transaction to be rolled back or "canceled."

General Format

```
ROLLBACK TRANSACTION
```

General Rules

The following rules describe how transaction management operates with Vision and relative files. For other file systems linked with the runtime, each system's native mechanism for transaction management is invoked. See the interface document for the specific file system for more details.

1. ROLLBACK locks the log file, checks its integrity, then writes a ROLLBACK notation to the log file and unlocks it.

2. When ROLLBACK is enabled in the FILE-CONTROL entry for a file, the record and file locking rules are extended for that file. Every record updated as part of a transaction is locked until that transaction is committed or rolled back. The ROLLBACK verb removes these locks. Record locks applied when reading the file are also kept until the end of the transaction.

3. During a transaction involving Vision or relative files, a CLOSE of a file that is locked, or that has locked or deleted records, is postponed until the transaction is committed or rolled back. If the same physical file is opened again within the transaction, even if the program is using a different logical file (different SELECT), the postponed CLOSE is canceled. Note that the mode of the original OPEN is retained. (For example, if the file were originally OPEN I-O, and if the CLOSE were canceled, then an OPEN OUTPUT on the same file within the same transaction would *not* recreate the file.) When the second OPEN is encountered, the file position is reset to the beginning so that a READ NEXT would read the first file in the record. CLOSE is handled in this special way so that record locks are held—these locks are necessary for rollback.

4. If the runtime system is killed by the user or encounters a fatal error prior to completing a transaction, an automatic rollback occurs.

5. Unless the **LOGGING** configuration variable is set to "0", file operations that occur in transactions are logged and recoverable regardless of whether the files have rollback capability.

6. Temporary files used for rollback are created in the working directory, or in the directory specified by the **LOG_DIR** configuration variable of the runtime.

7. The first write or rewrite on a sequential access mode file after a ROLLBACK TRANSACTION will be successful even if the primary key was written out of sequence, and even if the primary key on a rewrite does not match the last record read. No file error 22 will occur. This allows the program to continue where it left off after a rollback.

8. If the **STOP_RUN_ROLLBACK** configuration variable is set to 1, an implicit ROLLBACK occurs before a STOP RUN or before the end of the program.

SEARCH Statement

The SEARCH statement searches an indexed table for a specific table entry. The search may be sequential or binary (SEARCH or SEARCH ALL). The search terminates when either a match is found (first match), or when the entire table has been searched.

Note: This manual entry includes code examples and highlights for first-time users following the General Rules section.

General Format

Format 1

```
SEARCH table-name [ VARYING index-item ]

 [ AT END statement-1 ]

 { WHEN srch-cond {statement-2   }   } ...
                  {NEXT SENTENCE }

 [ END-SEARCH ]
```

Format 2

```
SEARCH ALL table-name

 [ AT END statement-1 ]

 WHEN { tbl-item   {IS EQUAL TO} value    }
      {            {IS =       }          }
      { cond-name                         }

      [ AND { tbl-item   {IS EQUAL TO} value } ] ...
            {            {IS =       }       }
            { cond-name                       }

            { statement-2   }
            { NEXT SENTENCE }

 [ END-SEARCH ]
```

Syntax Rules

1. *Table-name* is a data item that must contain an OCCURS clause including an INDEXED BY phrase. *Table-name* must *not* be subscripted in the SEARCH statement. In Format 2, *table-name* must also contain the KEY IS phrase in its OCCURS clause.

2. *Index-item* is a numeric integer data item or an index name. It may not be subscripted by the first index name in the INDEXED BY phrase in the OCCURS clause of *table-name*.

3. *Srch-cond* is a conditional expression.

4. *Statement-1* and *statement-2* are imperative statements.

5. *Value* may be a data item, a literal, or an arithmetic expression. It must be legal to compare *value* with *tbl-item*. No data item in *value* may be referenced in the KEY IS phrase in the OCCURS clause of *table-name*, nor may it be subscripted by the first index-name associated with *table-name*.

6. *Cond-name* is a condition-name (level 88) that must be defined as having only a single value. The condition-variable associated with *cond-name* must appear in the KEY IS phrase in the OCCURS clause of *table-name*.

7. *Tbl-item* must be subscripted by the first index-name associated with *table-name* along with other subscripts as required. It must be referenced in the KEY IS phrase in the OCCURS clause of *table-name*. *Tbl-item* may not be reference modified.

8. In Format 2, when a *tbl-item* or a *cond-name* is referenced, all preceding data-names in the KEY IS phrase in the OCCURS clause of *table-name* (or their associated condition-names) must also be referenced.

General Rules

Format 1

1. The Format 1 SEARCH statement searches a table serially starting with the current index setting.

   a. If the index-name associated with *table-name* contains a value that is higher than the highest occurrence number for *table-name*, the search terminates immediately. If the AT END phrase is specified, *statement-1* executes. Control then passes to the end of the SEARCH statement.

   b. If the index-name associated with *table-name* contains a valid occurrence number, the SEARCH statement evaluates the WHEN conditions (*srch-cond*) in the order they appear. If no condition is satisfied, the index-name associated with *table-name* is set to the next occurrence number. The evaluation process is then repeated. This process ends when a condition is satisfied or an occurrence number outside of the range of *table-name* is generated. In this second case, processing continues as in step (1a) above.

   c. When a *srch-cond* is satisfied, the SEARCH terminates and the associated *statement-2* executes (or control passes to the next sentence if NEXT SENTENCE is used). The index-name associated with *table-name* remains set at its current value. Control then passes to the end of the SEARCH statement.

2. If there is no VARYING phrase specified, the index-name used for the search is the first index-name in the INDEXED BY phrase associated with *table-name*. Other index-names associated with *table-name* remain unchanged.

3. If the VARYING phrase is specified, and *index-item* names an index-name associated with *table-name*, then that index-name is used for the search operation. If *index-name* names some other index-name or a numeric data item, that item is incremented by 1 every time the index-name associated with the search operation is incremented. The index-name specified in rule 2 is used for the search procedure.

Format 2

1. A Format 2 SEARCH performs a binary search of an ordered table. It yields predictable results only when:

   a. the data in the table has the same order as specified by the KEY IS phrase associated with table-name

   b. the contents of the keys in the WHEN phrase identify a unique table element

2. The initial value of the *table-name* index-name is ignored. It is varied in a non-linear manner by the SEARCH operation until the WHEN conditions are satisfied or the table has been searched.

3. If the WHEN phrase conditions are not satisfied for any index setting, control passes to the AT END phrase *statement-1*, if any, or to the end of the SEARCH statement. The setting of the *table-name* index-name is not predictable in this case.

4. If all of the WHEN phrase conditions are satisfied for an index setting, control passes either to the associated *statement-2* or to the next sentence, whichever is specified. The *table-name* index-name indicates the occurrence number that satisfied the conditions.

5. The index-name used for the search is the first index-name listed in the INDEXED BY phrase associated with *table-name*. Other index-names remain unchanged.

Code examples

Example 1:

In this example SEARCH is used to conduct a sequential search of the table for the first match. The index data item must be assigned an initial value by the program. Note that subsequent searches of the table for additional matches may be made if the value of the search index is saved after a match.

Assume the following table data item:

```
01  FRUIT-TREE-INVENTORY.
    05  FRUIT-TREE-TABLE
        OCCURS 100 TIMES
        INDEXED BY FTT-INDEX.
        10  FT-NAME    PIC X(25).
        10  FT-CODE    PIC X(5).
        10  FT-PRICE   PIC 9(5)V99.
        10  FT-COUNT   PIC 999.
*05 table name is specified by SEARCH
*OCCURS and INDEXED BY required for SEARCH
```

Assume that FRUIT-TREE-TABLE has been loaded.

```
*use SET to initialize the index
SET FTT-INDEX TO 1.
SEARCH FRUIT-TREE-TABLE
*handle no match in table
   AT END DISPLAY "Variety not found."
*test for match
   WHEN FT-NAME (FTT-INDEX) = TREE-NAME
*match found, perform action
      PERFORM DISPLAY-INVENTORY-ITEM
END-SEARCH.
```

Example 2:

In this example a WHEN clause is used in a sequential search to test for an "end of table" (AT END equivalent) condition. Note that when the table being searched is not full (has table elements at the end that have not been filled), searching the table into the unfilled space will give unpredictable results. You can search a partially filled table by determining the position of

the last valid entry in the table and then using a WHEN clause in the SEARCH statement to test for when the search process traverses past the last valid entry.

Assume the same table declaration as in example 1. Assume, also, that the program has verified the table entries and has saved the subscript value of the last valid entry in a variable named LAST-VALID-ENTRY.

```
*initialize the search index
SET FTT-INDEX TO 1.
SEARCH FRUIT-TREE-TABLE
*test for match
   WHEN FT-NAME (FTT-INDEX) = TREE-NAME
*match found, perform action
      PERFORM DISPLAY-INVENTORY-ITEM
*test for indexing into unfilled table space
   WHEN FTT-INDEX > LAST-VALID-ENTRY
*exit the SEARCH statement
      NEXT SENTENCE.

if ftt-index > last-valid-entry
      display " variety not found".
```

Example 3:

In this example SEARCH ALL is used to conduct a binary search of an ordered table. **Binary searches require sequential, ordered tables.** The table definition must include an ASCENDING or DESCENDING KEY clause. The search terminates upon first match, and there is no way to continue the search to find a second match. Binary searches are best suited to large tables (typically 50 records or more). When used to search large tables, the binary search method will, on average, find a table record much more quickly than will a sequential search. For example, a table containing 1000 records will need to perform no more than ten comparisons to find a match.

Assume the following table data item:

```
01  FRUIT-TREE-INVENTORY.
    05 FRUIT-TREE-TABLE
*OCCURS required for SEARCH
      OCCURS 100 TIMES
*ASCENDING/DESCENDING KEY required
*for SEARCH ALL
```

```
      ASCENDING KEY IS FT-NAME
*INDEXED BY required for SEARCH
      INDEXED BY FTT-INDEX.
      10  FT-NAME    PIC X(25).
      10  FT-CODE    PIC X(5).
      10  FT-PRICE   PIC 9(5)V99.
      10  FT-COUNT   PIC 999.
```

Assume the table has been loaded.

```
*FTT-INDEX is initialized by SEARCH ALL
SEARCH ALL FRUIT-TREE-TABLE
*Handle no match in table.
    AT END DISPLAY "Variety not found"
*Test for match
    WHEN FT-NAME (FTT-INDEX) = TREE-NAME
*Match found, perform action
    PERFORM DISPLAY-INVENTORY-ITEM
END-SEARCH.
```

Example 4:

This example demonstrates how to use SEARCH or SEARCH ALL to search multi-dimensional tables:

SEARCH is not, by itself, equipped to perform multi-dimensional table searches. One approach to accomplishing multi-dimensional table searches is to use SEARCH in conjunction with PERFORM/VARYING (as the following example will illustrate). When used together, SEARCH handles lookups at the innermost level (dimension) of the table structure and PERFORM/VARYING is used to manage stepping through the outer levels of the table.

Assume the following table data item:

```
01  TREE-INVENTORY.
    05  NURSERY-YARD                  |inventory location,
            OCCURS 10 TIMES            |"outer" table
            INDEXED BY YARD-IDX.
            10  TREE-TABLE            |tree type,
                   OCCURS 100 TIMES    |"inner" table
                   INDEXED BY TT-IDX.
                15  FT-NAME       PIC X(25).
                15  FT-CODE       PIC X(5).
```

```
               15  FT-PRICE      PIC 9(5)V99.
               15  FT-COUNT      PIC 999.
```

Assume the table has been loaded.

```
MOVE "N" TO TREE-FOUND.
PERFORM SEARCH-TREE-INVENTORY
*step through the outer table
   VARYING YARD-IDX FROM 1 BY 1
      UNTIL YARD-IDX > 10 OR TREE-FOUND = "Y".

IF TREE-FOUND = "N"            |note that this code
    PERFORM NO-TREE-FOUND.     |executes after the
END-IF.                        |search is complete
{ . . . }
SEARCH-TREE-INVENTORY.
   SET TT-IDX TO 1.
   SEARCH TREE-TABLE
      WHEN TREE-TABLE(YARD-IDX,TT-IDX) = TREE-NAME

*note that both the inner and outer table
*indexes are required
         PERFORM DISPLAY-INVENTORY-ITEM
         MOVE "Y" TO TREE-FOUND
   END-SEARCH.
```

If the inner table is ordered and large enough to benefit from a binary search, use SEARCH ALL.

Highlights for first-time users

General notes:

1. The table name identifier used in SEARCH must be the table name specified in the OCCURS phrase of the table declaration. You cannot use the 01 table label that starts the table declaration.

2. If END-SEARCH is used NEXT SENTENCE cannot be used. Where possible it is best to use the sentence terminator, END-SEARCH. Unintended logic errors are often introduced by the use of NEXT SENTENCE and are easily avoided by the use of END-SEARCH.

Notes regarding sequential searches (SEARCH):

1. A sequential search is conducted as follows:

   a. The search cycle begins by verifying that the value of the index data item falls within the range of the table size (the range is from 1 to the value specified in the OCCURS clause of the record definition).

   b. If the index value is valid, then each WHEN condition phrase is evaluated until either a match is found or until all WHEN conditions have been tested.

   c. If there is no match, the value of the index is incremented by one, validated (as in step a), and the WHEN condition evaluation cycle is repeated.

   d. Steps a - c iterate until either a match is found or the value of the index exceeds the table range, indicating that the entire table has been searched.

   e. If a match is found, the search terminates and the imperative statement associated with the WHEN clause is executed. Program execution then resumes immediately after the SEARCH statement. Note that the value of the index data item remains set to the value of the subscript of the matched entry.

   f. If the value of the search index ever becomes less than one or greater than the table size, the search terminates, the optional AT END statement, if present, is executed, and program execution continues immediately after the SEARCH statement.

2. The index data item named in the INDEXED BY clause is used to index the table in the sequential search and must be explicitly initialized in the program. Use SET to assign the initial value. When the search results in a match, the index data item remains set to the table subscript of the matching entry. Saving or preserving this value makes it possible to make another search of the table for a subsequent match.

   Initializing the index data item:

a. If the entire table is to be searched, the index data item should be assigned, using SET, the value 1, thereby starting the search with the first record.

b. If the search is to begin with an entry other than the first, then the index data item should be assigned the value of the position of the first table entry to be checked. For example, to start the search at table entry 10, assign the value 10 to the index data item.

c. If, after a search finds a match, you want to make an additional search of the table to find a subsequent match, the value of the index data item should be preserved and then reassigned so that the next search begins at 1 + *index-item*.

3. When searching tables that are not full (do not contain valid entries for every occurrence in the table), use a WHEN clause to test for the actual end-of-table condition. If the search is allowed to proceed into the unused portion of the table, garbage values in the unfilled table space will give unpredictable results. See code example 2.

4. The relational match conditions associated with each WHEN clause may be connected with the logical connectors AND or OR thereby specifying multiple or alternate match conditions. For example:

```
WHEN NAME = SEARCH-NAME OR SIZE < MAX-SIZE
```

5. Use of the VARYING phrase: The VARYING phrase allows alternate or multiple indexes to be incremented by the search loop. If VARYING is omitted, the first index-item defined in the INDEXED BY phrase of the OCCURS clause (for the table) is incremented.

   If the VARYING phrase is included, the index item named after VARYING is incremented, as well as the first named index in the INDEXED BY phrase, with one exception. If the index named after VARYING is also named in the INDEXED BY phrase, then it is the only index incremented.

Notes regarding binary searches (SEARCH ALL):

1. The binary search is conducted as follows:

   a. The search begins at the midpoint of the table (for example, 50 of 100) and compares the value of the table entry with the search item to determine if there is a match.

b. If there is no match, SEARCH determines whether the search item is logically located in the upper or lower half of the table (0-49, or 51-100).

c. SEARCH then finds the midpoint of the half that logically contains the search item and determines if the table element at the midpoint matches the search item.

d. If there is no match, SEARCH again determines whether the search item is logically located in the upper or lower half of the remaining range.

e. This process iterates until the search item is found or until it is determined that the table does not contain the search item (the remaining table range becomes null).

f. If at any time a match is found, the search immediately terminates and the imperative statement associated with the WHEN clause is executed. Program execution then resumes immediately after the SEARCH statement.

2. Binary searches require sequential, ordered tables (via use of the ASCENDING/DESCENDING KEY phrase). The table must be ordered as specified by the KEY IS phrase of the table definition.

3. The matching conditions of the WHEN clause must identify a unique table entry.

4. Binary searches are best suited to large tables (typically 50 records or more) where the binary search algorithm significantly reduces the average number of lookups per match.

5. Unlike a sequential search, the binary search format permits only one WHEN clause.

6. Because only one WHEN clause is permitted and because the index value is automatically set by the program, it is not possible to SEARCH partially full tables.

7. The SEARCH ALL match conditions are very restrictive. Match condition evaluation is restricted to evaluation of a condition-name, which can represent only a single value (no range or sequence of values permitted), *or* a condition which tests for equality.

8. The table data item and the index must be on the left side of the condition statement.
9. Multiple condition tests are permitted but can be connected only with an AND (no OR).
10. Any table item or condition-name referenced must be named in the KEY IS phrase of the OCCURS clause of the table definition.
11. The VARYING option is not permitted.
12. When a match is found, the index retains the value of the table subscript of the matched entry. If no match is found the value of the index is unpredictable.

SEND Statement

The SEND statement sends a message to other threads.

General Format

```
SEND src-item TO { { THREAD dest-thread } ... }
                 { LAST THREAD                 }
                 { ALL THREADS                 }
```

Syntax Rules

1. *Src-item* is a literal or data item.
2. *Dest-thread* is a USAGE HANDLE or HANDLE OF THREAD data item.

General Rules

1. The SEND statement sends a message containing the data in *src-item* to one or more threads. Which threads receive the message depends on the following:
   a. THREAD *dest-thread* causes the message to be sent to the thread identified by *dest-thread*. More than one *dest-thread* can be specified.

b. LAST THREAD causes the message to be sent to the *last* thread (see section 6.8.1, Book 1, *ACUCOBOL-GT User's Guide* for a discussion of the *last* thread).

c. ALL THREADS causes the message to be sent to all currently existing threads, except the sending thread.

2. The size of the message is equal to the size of *src-item*.

3. The THREAD *dest-thread* and LAST THREAD options create a *directed* message. Directed messages are sent to the specified threads or last thread and are guaranteed to be delivered to the specified threads. Directed messages are held in a queue. If there is not enough space in the queue to place the message (because of other messages that have not yet been received), the sending thread suspends until space becomes available in the queue. Messages are received in the order sent. See the listing for **MESSAGE_QUEUE_SIZE** runtime configuration variable located in Appendix H for options on setting the queue size.

4. The ALL THREADS option creates a *broadcast* message. Broadcast messages can be picked up by any thread. Receiving a broadcast message does not remove it from the message queue, it remains queued to be received by other threads. Broadcast messages are removed from the queue when either

   a. all threads have received it, or

   b. there is not enough space to hold the next broadcast message. In this case, broadcast messages are removed from the queue (oldest first) until there is enough space to place the new message in the queue. This allows the queue to empty when there are threads that never look for messages. It also means that a broadcast message may not be delivered to a particular thread if other broadcast messages are sent before the first is received. To avoid this problem, use broadcast messages sparingly.

5. Broadcast messages can be missed under certain circumstances. To track which messages a thread has read, each thread remembers the number of the last broadcast message it has read. Messages are numbered sequentially starting at one. If a thread skips an earlier broadcast message (because the message does not meet the thread's delivery requirements) and then receives a later broadcast message, the

earlier message will never be received because it has an earlier message number. For example, MESSAGE-1 will be missed in the following sequence of events:

```
SEND MESSAGE-1 TO ALL THREADS     (in thread A)
SEND MESSAGE-2 TO ALL THREADS     (in thread B)
RECEIVE MSG-2 FROM THREAD B       (in thread C)
RECEIVE MSG-1 FROM ANY THREAD     (in thread C)
```

The last RECEIVE in this example does not receive MESSAGE-1 because it was sent earlier than the last broadcast message it received (MESSAGE-2 in this case). This is caused by the first RECEIVE which picked up a broadcast message while asking for messages from a particular thread. To avoid this, pick up broadcast messages with the RECEIVE FROM ANY THREAD phrase. This will prevent you from skipping a message.

SET Statement

The SET statement sets the values of various types of data items, allows you to control the *current* and *active* windows, and allows you to set the priority of a thread.

General Format

Format 1

```
SET {result} ... TO value
```

Format 2

```
SET {result} ... {UP  } BY value
                 {DOWN}
```

Format 3

```
SET { {cond-name} ... TO {TRUE } } ...
                         {FALSE}
```

Format 4

```
SET { {switch-name} ... TO {ON } } ...
                           {OFF}
```

Format 5

```
SET FILE-PREFIX TO file-prefix
```

Format 6

```
SET {CONFIGURATION} { env-name TO env-value } ...
    {ENVIRONMENT  }
```

Format 7

```
SET pointer TO { ADDRESS OF data-item }
               { NULL                 }
```

Format 8

```
SET result-item TO SIZE OF data-item
```

Format 9

```
SET ADDRESS OF linkage-item TO { pointer             }
                               { ADDRESS OF data-item}
                               { NULL                }
```

Format 10

```
SET {INPUT       } WINDOW TO window-1
    {INPUT-OUTPUT}
    {I-O         }
    {OUTPUT      }
```

Format 11

```
SET {handle-1} ... TO HANDLE OF {screen-1       }
                                {CONTROL ID id-1}
```

Format 12

```
SET THREAD {thread-id} PRIORITY TO priority
```

Format 13

```
SET EXCEPTION {VALUE } { exc-value TO {ITEM-HELP          } } ...
              {VALUES}                {HELP-CURSOR        }
                                      {CUT-SELECTION      }
                                      {COPY-SELECTION     }
                                      {PASTE-SELECTION    }
                                      {DELETE-SELECTION   }
                                      {UNDO               }
```

{SELECT-ALL-SELECTION}

Syntax Rules

1. *Result* is a numeric data item or index name.
2. *Value* is a numeric literal, a numeric data item, or an index name.
3. *File-prefix* is a nonnumeric literal or alphanumeric data item.
4. *Cond-name* is any condition-name (level 88 item). If the FALSE option is used, then *cond-name* must have a WHEN SET TO FALSE phrase in its definition.
5. *Switch-name* must be a mnemonic name associated with an external switch in the SPECIAL-NAMES section of the Environment Division.
6. *Env-name* is a nonnumeric literal or data item.
7. *Env-value* is a USAGE DISPLAY numeric or nonnumeric literal or data item. If numeric, it must be an integer.
8. CONFIGURATION and ENVIRONMENT are equivalent.
9. *Pointer* must be a data item with USAGE POINTER.
10. *Result-item* must be a numeric data item.
11. *Linkage-item* must be declared in the Linkage section.
12. *Window-1* is a USAGE HANDLE or PIC X(10) data item that refers to a floating window or the main application window.
13. *Handle-1* is a USAGE HANDLE data item. When the control is an ActiveX, COM, or .NET control, *handle-1* must be a *typed* handle that matches the control; i.e., *handle-1* must be declared with the "USAGE HANDLE OF *control-type*" syntax. See **section 5.7.1.8, "USAGE clause."**
14. *Screen-1* must refer to an elementary Screen Section item that describes a graphical control.
15. *Id-1* and *Priority* are numeric literals or data items.
16. *Thread-id* is a USAGE HANDLE or HANDLE OF THREAD data item.

17. *Exc-value* is an integer literal or data item.

General Rules

Format 1

Each *result* is set to *value*. This assignment is done such that the numeric values of *result* and *value* will be the same.

Format 2

Value is either added to (UP BY) or subtracted from (DOWN BY) each *result* item. No size error checking is done.

Format 3

1. When the TRUE phrase is used, the literal in the VALUE clause for *cond-name* is moved to its associated condition-variable. If the VALUE clause contains more than one literal, the first one is used.
2. When the FALSE phrase is used, the literal defined in the WHEN SET TO FALSE phrase of *cond-name* is moved to its associated condition-variable.

Format 4

Format 4 of the SET statement alters the on/off status of external switches. These switches are initially "off" unless otherwise specified when the program is run.

Format 5

1. The FILE-PREFIX is a special register maintained by ACUCOBOL-GT to aid in translating COBOL ASSIGN names to actual file names on the host computer. A complete description of its function is located in section 2.8, "File Name Interpretation," of the *ACUCOBOL-GT User's Guide*.
2. A Format 5 SET statement is equivalent to this Format 6 SET statement:

   ```
   SET ENVIRONMENT "FILE-PREFIX" TO file-prefix
   ```

Format 6

1. ACUCOBOL-GT maintains a set of *configuration variables* that can affect various aspects of the runtime system. These variables can be initially set in the ACUCOBOL-GT runtime configuration file described in Chapter 2 of the *User's Guide*. The Format 6 SET statement can be used to modify these values at runtime.

2. *Env-name* is the name of the configuration variable to set. In it, lower-case characters are treated as upper case, and underscores are treated as hyphens. The first space character delimits the name. *Env-name* may specify either the literal name of the variable or a data-item whose value is the name of the variable. If you specify the actual name of the variable, such as COMPRESS-FILES, then you must enclose the name in quotes. *Env-value* is the value to set the variable to. If it is a numeric data item, then it is treated as if it were redefined as an alphanumeric data item.

3. If *env-name* does not match the name of one of the runtime system's configuration variables, then *env-name* and *env-value* are placed in the runtime system's local environment. These entries are used to do file name translations—see the *ACUCOBOL-GT User's Guide*, **section 2.9, "File Name Interpretation."**

4. The complete list of environment variables used by ACUCOBOL-GT can be found in Appendix H, Book 4, *Appendices*.

Format 7

If the ADDRESS OF option is used, then the address of *data-item* is stored in *pointer*. If the NULL option is used, then *pointer* is set to point to no data item.

Note that the "-Zm" compiler option Causes the compiler to generate code that tells the runtime the size of a data item specified in the SET statement. See Book 1, **Section 2.2.16, "Miscellaneous Options"** for details on the -Zm option.

Format 8

The number of standard character positions occupied by *data-item* is stored in *result-item*.

Format 9

1. If *pointer* is specified, then the address of the *linkage-item* is set to *pointer*. If the ADDRESS OF option is used, then the address of the *linkage-item* is set to the address of *data-item*. If the NULL option is used, then the address of the *linkage-item* is set to point to no data item.
2. The level of *linkage-item* must be either 01 or 77.
3. If the *linkage-item* is not listed in the PROCEDURE DIVISION USING phrase and is referenced before the SET ADDRESS OF statement, then the runtime will abort with the message, "Use of a LINKAGE data item not passed by the caller".

Format 9 is helpful if you want to allocate a sizable piece of memory for temporary use, access this memory via a COBOL table, and then free the memory. For example:

- Use the library routine M$ALLOC to allocate the memory.
- SET the address of a Linkage section table to the pointer returned from M$ALLOC.
- Complete the desired procedures.
- Free the memory with M$FREE.

The code sample that follows shows how Format 9 works.

```
identification division.

program-id.     sample-program.

data division.

working-storage section.

linkage section.

* item-a and ptr-a are passed in by the calling
* program

    01 item-a    pic x(10).
```

```
* ptr-a is set to the address of item-a
* in the calling program

    01 ptr-a   usage pointer.

* item-b is used in the SET Statement.
* It is not passed in by the calling program.

    01 item-b   pic x(10).

procedure division using item-a, ptr-a.
main-logic.

* Assuming item-a has a value of "ABCDEFGHIJ",
* and ptr-a points to item-a,
* the following will display "ABCDEFGHIJ" three
* times

    display item-a.

* "ABCDEFGHIJ" is displayed

    set address of item-b to ptr-a.
    display item-b.

* "ABCDEFGHIJ" is displayed

    set address of item-b to address of item-a.
    display item-b.

* "ABCDEFGHIJ" is displayed

    stop run.
```

Format 10

1. Format 10 of the SET verb makes *window-1* the current, or current and active window. The *current* window is the window to which DISPLAY statements refer. The *active* window is the window that is highlighted and the one to which user input is directed. *Window-1* must be a handle to a valid floating window. If *window-1* does not refer to a valid floating window, the SET statement has no effect.

2. INPUT, INPUT-OUTPUT, and I-O are synonymous. They cause *window-1* to become both the current and active window.

3. OUTPUT causes *window-1* to become the current window.

Format 11

A Format 11 SET statement retrieves the handle to the control described by *screen-1* or *id-1* and stores it in *handle-1*. *Id-1* must specify a value greater than zero. If a matching control is found, *handle-1* is set to the handle of that control. If no matching control is found, *handle-1* is set to NULL. If more than one control has a matching ID, then *handle-1* is arbitrarily set to one of those controls. Note that the handle can be used in any statement that can use control handles. One reason you might want this handle is if you need to pass a control to a subprogram. You cannot pass Screen Section names to subprograms, but you can pass the handle instead.

Format 12

1. A Format 12 SET statement sets the execution priority of a thread. Execution switches between threads at various points in the program. Each opportunity to change the active thread is called a *switch point*. The execution priority determines which thread gets control at each switch point.

2. The execution priority is an integer. The higher the priority, the more often that thread gets control at a switch point. By default, threads start with a priority value of 100. Threads receive control in proportion to their priority. Thus, a thread with a priority of 50 gains control half as often as a thread with a priority of 100. Of course, if a thread is paused for any reason (waiting for input, for example), then it does not gain control.

3. If *thread-id* is specified, then the priority for the thread identified by *thread-id* is set to *priority*. Otherwise, the current thread's priority is set to *priority*. If *thread-id* does not correspond to an existing thread, then the SET statement has no effect.

4. The minimum priority for a thread is "1". The maximum is 32767.

Format 13

A Format 13 SET statement associates the exception value specified in *exc-value* with an automated action that the runtime can perform. Any keystroke, menu item, or control that produces the *exc-value* exception value will automatically cause the associated action to be performed (you do not have to code the action; it happens automatically). If the runtime handles the exception in this way, then the exception is not passed on to the COBOL program.

The ITEM-HELP action produces context-sensitive help for the control with the current input focus. The HELP-CURSOR action places the mouse into help mode. For a description of the ITEM-HELP and HELP-CURSOR actions, see section 10.4, Book 2, *ACUCOBOL-GT User Interface Programming*.

The remaining actions take effect if the current control is an entry field (otherwise, they have no effect). These actions cause the entry field to do the following:

CUT-SELECTION	Cuts the current selection to the clipboard
COPY-SELECTION	Copies the current selection to the clipboard
PASTE-SELECTION	Pastes the clipboard into the entry field at the current location (replaces any existing selection)
DELETE-SELECTION	Deletes the current selection
UNDO	Undoes the last change
SELECT-ALL-SELECTION	Selects all the text in the entry field. In a multi-line entry field, this includes the text in all lines.

The cut, copy, paste, delete, and undo effects are accomplished automatically via the ACTION property of entry fields. Usually, you will want to assign these exception values to various menu items and toolbar push buttons. When you are setting up a push button to correspond to one of these actions,

you should ensure that you make the push button a SELF-ACT button (otherwise the act of pushing the button makes the button the current control, not the entry field).

SORT Statement

The SORT statement sorts records according to selected key fields. Record content can be modified before and after the actual sort process using INPUT PROCEDURE and OUTPUT PROCEDURE.

ACUCOBOL-GT also including a sorting utility called **AcuSort**. See chapter 3 of the *ACUCOBOL-GT User's Guide* for details on this utility. There is also a runtime configuration variable that instructs the runtime to use the system's Quicksort algorithm (if present) instead of the built-in algorithm specified by the SORT statement. See the **USE_SYSTEM_QSORT** variable in Appendix H of the *ACUCOBOL-GT Appendices* manual.

Note: This manual entry includes code examples and highlights for first-time users following the General Rules section. In the highlights list, item four discusses ways to improve SORT performance.

General Format

```
SORT sort-file

  { KEY AREA IS key-table }
  { ON {ASCENDING } KEY {key-name} } ...
       {DESCENDING}

  [ WITH DUPLICATES IN ORDER ]

  [ COLLATING SEQUENCE IS alpha-name ]

  { INPUT PROCEDURE IS proc-name  }
  { USING {in-file} ...           }

  { OUTPUT PROCEDURE IS proc-name }
  { GIVING {out-file} ...         }
```

Note that *proc-name* has the following format:

```
start-proc [ {THRU   } end-proc ]
             {THROUGH}
```

Syntax Rules

1. *Sort-file* names a sort file described by an SD entry in the Data Division.

2. *Key-table* must name a data item that is *not* located in the record for sort-file. *Key-table* may not be subordinate to an OCCURS clause, nor may it be reference modified.

3. *Key-table* must reference a data item whose size is an even multiple of 7. *Key-table* is processed as if it had the following structure:

   ```
   01  KEY-TABLE.
       03  SORT-KEY OCCURS N TIMES.
           05  KEY-ASCENDING   PIC X  COMP-X.
           05  KEY-TYPE        PIC X  COMP-X.
           05  KEY-OFFSET      PIC XX COMP-X.
           05  KEY-SIZE        PIC XX COMP-X.
           05  KEY-DIGITS      PIC X  COMP-X.
   ```

   Typically, programs will declare *key-table* with a similar format.

4. *Key-name* is a data item in the record description associated with *sort-file*. It may not be subordinate to an OCCURS clause, nor may it be a group item containing variable occurrence data items. It may not be reference modified. The maximum number of keys allowed is 23.

5. *Alpha-name* is an alphabet-name defined in the SPECIAL-NAMES paragraph of the Environment Division.

6. *In-file* and *out-file* are files described by FD entries in the Data Division. They may not be sort files. The maximum number of input and ouput files is 25.

7. *Start-proc* and *end-proc* are paragraph or section names in the Procedure Division.

8. A SORT statement may not appear in Declaratives or in the input or output procedure of a SORT or MERGE statement.

9. If *sort-file* contains variable length records, *in-file* records must not be smaller than the smallest record in *sort-file* nor larger than the largest. If *sort-file* contains fixed length records, *in-file* records may not be larger than the size of *sort-file's* records.

10. If *out-file* contains variable length records, *sort-file* records must not be smaller than the smallest record in *out-file* nor larger than the largest. If *out-file* contains fixed length records, *sort-file* records may not be larger than the size of *out-files* records.

11. If *sort-file* contains more than one record description, *key-name* need appear in only one of them. The character positions referenced by *key-name* are used as the key for all the file's records.

12. If *out-file* is an indexed file, the first *key-name* must be ASCENDING and must specify the same character positions in its record as the primary record key for *out-file*.

13. THRU is an abbreviation for THROUGH.

General Rules

1. The SORT statement sorts records received from the INPUT PROCEDURE or found in the *in-files*. It then either makes these sorted records available to the OUTPUT PROCEDURE or writes them to each *out-file*.

2. Sort records must be at least six bytes in size.

3. If *sort-file* contains fixed length records, any shorter *in-file* records are space-filled on the right to match the record size.

4. If *out-file* contains fixed length records, any shorter *sort-file* records are space-filled on the right to match the record size.

5. The first *key-name* is the major key, and the next *key-name* is the next most significant key. This pattern continues for each *key-name* specified.

6. The ASCENDING phrase specifies that key values are to be ordered from lowest to highest. The DESCENDING phrase specifies the reverse ordering. Once ASCENDING or DESCENDING is specified, it applies to each *key-name* until another ASCENDING or DESCENDING adjective is encountered.

7. Use the KEY AREA option when you do not know the specifics of the sort key until the program is run. You can use this to allow users to enter sort key specifications, typically in conjunction with some form of data dictionary.

8. Your program must fill in a table of information that describes the sort keys. This table, *key-table*, should have the format described by Syntax Rule 3 above. The number of sort keys is determined by the number of occurrences in the table. The keys are listed in order of precedence: table entry 1 describes the highest precedence key, table entry 2 the second highest, and so on. If you need to process a variable number of keys, use a variable-size table (by using OCCURS DEPENDING ON).

9. For each key, you must specify the following information:

KEY-ASCENDING:	This should be 0 or 1. Enter 1 to have an ascending sort sequence, 0 for descending.
KEY-TYPE:	Describes the underlying data format. The allowed values are listed in the next rule.
KEY-OFFSET:	Describes the distance (in standard character positions) from the beginning of the sort record to the beginning of the key field. The first field in a sort record is at offset 0.
KEY-SIZE:	Describes the size of the key field in standard character positions.
KEY-DIGITS:	This is used only for numeric keys. It describes the number of digits contained in the key (counting digits on both sides of the decimal point).

10. The KEY-TYPE field uses a code to describe the type and internal storage format of the data item. Select from the following values:

0	Numeric edited
1	Unsigned numeric (DISPLAY)
2	Signed numeric (DISPLAY, trailing separate)
3	Signed numeric (DISPLAY, trailing combined)

4	Signed numeric (DISPLAY, leading separate)
5	Signed numeric (DISPLAY, leading combined)
6	Signed COMP-2
7	Unsigned COMP-2
8	Unsigned COMP-3
9	Signed COMP-3
10	COMP-6
11	Signed binary (COMP-1, COMP-4, COMP-X)
12	Unsigned binary (COMP-1, COMP-4, COMP-X)
13	Signed native (COMP-5, COMP-N)
14	Unsigned native (COMP-5, COMP-N)
15	Floating point (FLOAT, DOUBLE)
16	Alphanumeric
17	Alphanumeric (justified)
18	Alphabetic
19	Alphabetic (justified)
20	Alphanumeric edited
21	Not used
22	Group

This coding is the same one used by the C interface, and is also used by Acu4GL to interface to relational DBMSs. When specifying the key type, you may safely use “alphanumeric” for all nonnumeric keys. (The sort rules are the same for each of these types). For numeric data, however, you must specify the correct type or you may get sorting errors.

11. The results are undefined if you provide invalid data in the *key-table*. If you fail to specify any keys (by specifying a table whose size is zero), you receive a file error on *sort-file*. Under the default file status codes, this is file error 94 with a secondary status of 63.

12. For nonnumeric keys, the COLLATING SEQUENCE phrase establishes the ordering. If this phrase is omitted, the NATIVE collating sequence is used. For numeric keys, the ordering is specified by the algebraic value of the key.

13. The DUPLICATES phrase affects the return order for records whose *key-name* values are equal.

    a. When there is a USING phrase, the return order is the same as the order of appearance of *in-file* names in the SORT statement. Within a given *in-file*, the order is that in which the records are accessed from that file.

    b. When there is an INPUT PROCEDURE, the return order is the same as the order in which records were released. If the DUPLICATES phrase is not used, the return order for records with equal key values is unpredictable.

14. The execution of a SORT statement consists of three distinct phases. These are:

    a. Records are made available to the *sort-file*. This is achieved either by executing RELEASE statements in the input procedure or by implicit execution of READ statements for each *in-file*. When this phase starts, *in-file* must not be open. When it finishes, *in-file* will not be open.

    b. The *sort-file* is sequenced according to the KEY phrase and the DUPLICATES clause. No processing of *in-files* or *out-files* takes place during this phase.

    c. The records in *sort-file* are made available in sorted order. The sorted records are either written to the *out-files* or are made available to an output routine through execution of a RETURN statement. When this phase starts, *out-file* must not be open. When it finishes, *out-file* will be closed.

15. If the INPUT PROCEDURE phrase is used, the named procedure is executed by the SORT statement according to the rules for the PERFORM verb. This procedure must make records available to the input phase of the sort operation by executing RELEASE statements. When this procedure returns, the sort operation proceeds to the sequencing phase. The range of the input procedure may not cause the execution of a MERGE, RETURN, or SORT statement.

16. If the USING phrase is specified, all records in each *in-file* are transferred to *sort-file*. For each *in-file*, the following actions occur:

    a. The file is opened as if it were the object of an OPEN INPUT statement with no options.

    b. The records are obtained and released to the sort operation. Each record is obtained as if a READ statement with the NEXT and AT END phrases had been executed. For relative files, the RELATIVE KEY data item is undefined at the end of this phase.

    c. The file is closed as if it were the object of a CLOSE statement with no options. This occurs prior to the sequencing of *sort-file*.

    These implicit functions are performed such that any associated USE procedures are executed. These USE procedures must not access *in-file* or its record area.

17. If an output procedure is specified, control passes to it after the *sort-file* has been sequenced. Control passes to the output procedure according to the rules of the PERFORM statement. The output procedure must execute RETURN statements to retrieve the sorted records. When the output procedure returns, the SORT statement terminates and control passes to the next executable statement. The range of the output procedure must not execute any MERGE, RELEASE, or SORT statements.

18. If the GIVING phrase is used, all the sorted records are written to each *out-file*. For each of these files, the following steps occur:

    a. *Out-file* is opened as if it were the object of an OPEN OUTPUT statement with no options.

    b. The sorted records are returned and written to the file. The records are written as if a WRITE statement without any options had been executed. For a relative file, the value of the RELATIVE KEY data item is updated to reflect the record number written.

    c. The file is closed as if it were the object of a CLOSE statement without any options.

    These implicit functions are performed such that any associated USE procedures are executed. Such a USE procedure may not refer to *out-file* or its record area. On the first attempt to write beyond the externally

defined boundaries of the file, any applicable USE procedure is executed. If control is returned from that USE procedure, or no USE procedure is applicable, the processing of that *out-file* is terminated.

19. If the SORT statement is in a fixed segment, the range of any input and output procedures must be contained completely in the fixed segments and no more than one independent segment. If the MERGE statement is in an independent segment, the range must be completely contained in the fixed segments and the same independent segment.

20. The SORT statement updates the value of the *sort-file's* FILE STATUS data item.

21. Only one SORT may be active at a time. See also "CANCEL SORT."

22. If a SORT statement is executed in a wrong context, the runtime displays the error "Illegal SORT." This error belongs to the class of "intermediate" runtime errors that, upon occurrence, call installed error procedures. See Book 4, *Appendices*, Appendix I "Library Routines," **CBL_ERROR_PROC** for details.

23. For compatibility with other COBOLs, ACUCOBOL-GT includes special registers known as SORT-RETURN and SORT-MESSAGE. SORT-RETURN can be used for two purposes.

    a. To determine the status of a SORT that's just finished. You can determine the success or failure of a SORT by examining this variable after the SORT returns. A value of "0" indicates success, and a non-zero value indicates failure.

    b. To interrupt a SORT that is currently running. By setting this variable in an input or output procedure, you stop SORT processing immediately after the next RELEASE or RETURN statement is performed. By setting this variable in a DECLARATIVES paragraph (if you are not using input or output procedures), you stop SORT processing immediately after the next implicit RELEASE or RETURN is performed.

    The special register SORT-RETURN is of type SIGNED-INT. Please note that this register is primarily for compatibility purposes, and there are better ways to perform these functions in ACUCOBOL-GT. For instance, to get status on a SORT, use the FILE STATUS variable of the

SORT file. This gives more information than just success or failure. And if you are using input procedures, you can halt a SORT more simply by returning from the procedure as if you had reached the end of file.

When compiling for IBM compatibility ("-Cv"), the SORT-MESSAGE behaves just like it is declared in every program, for example:

```
01  SORT-MESSAGE PIC X(8) EXTERNAL.
```

This variable is used in mainframe environments to help control the SORT operation. In ACUCOBOL-GT, the variable has no particular effect.

24. The SORT statement can be used to sort elements of a working-storage table. The syntax is:

```
SORT data-name-2 [ ON ASCENDING/DESCENDING KEY
data-name-1 ... ]
   [ WITH DUPLICATES IN ORDER ]
   [ COLLATING SEQUENCE IS alphabet-name ]
```

Code examples

Example 1:

```
SORT PRODUCT-SORT-FILE             |temporary SD file
   ON ASCENDING KEY MODEL-TYPE,    |major sort key
                    MODEL-NUMBER   |minor sort key
   USING ATHLETIC-SHOES-LIST,      |input data file
         DRESS-SHOES-LIST          |input data file
   GIVING PRODUCT-LIST.            |permanent output data file
```

Example 2

(An extended version of this example appears after the Highlights for First-Time Users section.):

```
SORT PRODUCT-SORT-FILE                |temporary SD file
*duplicates sorted in the order acquired
   ON ASCENDING  KEY MODEL-TYPE       |major sort key
   ON DESCENDING KEY MODEL-NUMBER     |minor sort key
   WITH DUPLICATES IN ORDER
   INPUT  PROCEDURE IS WEED-PRODUCT-LIST
   OUTPUT PROCEDURE IS UPDATE-PRODUCT-LIST.
```

Highlights for first-time users

1. SORT is used to order records according to a set of key fields (sort keys). Records may be stored in sequential, relative, or indexed files, or records may be acquired by use of an INPUT PROCEDURE. Once ordered, the record set may be further processed by use of an OUTPUT PROCEDURE, or the records may be written directly to the named output file(s).

2. SORT is most often used to order records stored in disk files. However, by using an INPUT PROCEDURE you can acquire records from other input sources such as output from batch processes, internal application data structures, or screen input.

3. SORT creates a special temporary disk file (the sort file) as a work space for collecting, sorting, and holding ordered records. The sort file is defined by an SD entry in the DATA DIVISION. The sort file record definition must immediately follow the SD entry and must include definitions for each sort key used, except when the KEY AREA phrase is used. You can place temporary files used by the SORT verb in a specified directory—see the **SORT_DIR** configuration variable in Appendix H, Book 4. The sort file is removed when the SORT statement completes.

4. ***Runtime performance:*** Most SORT procedures involve the reading, sorting, and writing of records stored in disk files. These disk I/O processes can be relatively slow and, therefore, the SORT process can take a lot of time. However, you can tune performance. To get the best runtime performance, give the process as much memory and as many temporary files as possible, without wasting resources or adversely affecting other processes running on the system. You can use the runtime configuration variables **SORT_MEMORY** and **SORT_FILES** to specify the amount of memory and the number of temporary files available to the process. The default values are relatively small. Determining the optimal values depends on the number and size of the records being sorted, the amount of available memory, and the needs of other processes on the system. Some experimentation may be necessary.

   By default, the SORT routine uses a built-in sort function. Alternatively, if your system has a qsort() function, you can specify its use by setting the runtime configuration variable called "**USE_SYSTEM_QSORT**" to

the value of "1". Some systems have qsort() functions that perform better than the built-in function. Consider experimenting with this variable to determine if this option yields better performance on your system. Pay particular attention to the number of comparisons done during the sort, which can be seen in the runtime trace output.

5. The three basic steps of the SORT procedure are:

   a. Acquiring and placing the records to be sorted in the sort file:

      When the USING phrase is used, SORT opens each named input file, reads the data records, one at a time, into the sort file and closes the input file. Input files must not be open before the SORT statement begins.

      When an INPUT PROCEDURE is used the RELEASE verb is used to pass records to the sort file. If records are acquired from disk files, it is the responsibility of the input procedure to open, read, process, RELEASE each individual record, and close the files. For more information, see the **RELEASE Statement**.

   b. Sorting the records:

      Using the sort file, and a set of temporary files, records are sorted according to the key phrase, the DUPLICATES clause, and the COLLATING clause. See also the configuration variables in Appendix H, Book 4, under **SORT_DIR**, **SORT_MEMORY**.

   c. Disposition of the sorted records:

      When the GIVING phrase is used, the sorted records are written to the named permanent output file(s).

      When an OUTPUT PROCEDURE is used, the sorted records are made available to the output procedure for processing and writing to a permanent file(s). The output procedure uses the verb RETURN to acquire the ordered records from the sort file. It is the responsibility of the output procedure to open, write, and close the output file(s). For more information see the description of the **RETURN Statement**.

6. A SORT statement may not appear in a DECLARATIVES section or in an INPUT or OUTPUT PROCEDURE that is part of a SORT statement (nesting of SORT statements is not permitted).

7. The KEY AREA phrase is a means for defining the sort keys at runtime, as the application is running. When you use KEY AREA, it is not required that the sort file record descriptor contain entries for potential sort keys. Definition of the sort key(s) in the sort file is handled internally by the SORT routine using the key table. See syntax rules 2 and 3 and general rules 6 through 10.

8. If the KEY AREA phrase is not used, the sort keys must be defined in the record description of the sort file.

9. Use of INPUT PROCEDURE or OUTPUT PROCEDURE requires that all file I/O operations and record disposition be handled by the input or output procedure. This means that the input and output procedures must explicitly perform the OPEN, READ, RELEASE (input), RETURN (output), WRITE, and CLOSE actions. As with any I/O management, the procedure should consider and account for the handling of all I/O related errors.

10. Use the DUPLICATES phrase when you want duplicate records to be sequenced in the same order that they are read in or RELEASEd. Duplicate records are those that have identical key values. In the absence of the DUPLICATES phrase sequencing of duplicate records is not predictable (see General Rule 12).

11. Use the COLLATING SEQUENCE phrase to alter the ordering of nonnumeric keys. The named collating sequence must be defined in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. In the SPECIAL-NAMES paragraph the user may define a unique character order, or the user may select one of the four predefined character sequences: STANDARD-1, STANDARD-2, NATIVE, and EBCDIC. See **section 4.2.3, "Special-Names Paragraph."**

    If no COLLATING SEQUENCE phrase is used, the default collating sequence is used. The default collating sequence is whatever is native to the operating system (usually the same as the predefined type NATIVE).

12. Use the STATUS variable to hold the execution status of the SORT operation. The status variable is named in the SELECT/ASSIGN phrase of the FILE-CONTROL paragraph of the INPUT-OUTPUT SECTION. See **section 4.2.3, "Special-Names Paragraph."**

    For a complete list and description of file status codes, see Appendix E, Book 4.

13. To specify the disk directory in which SORT will place any temporary files, set the **SORT_DIR** runtime configuration variable, located in the runtime configuration file.

Extended version of code example 2:

For simplicity, only one input file will be used.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  SAMPLE-FILE-SORT.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
*SORT input file
SELECT ATHLETIC-SHOES-LIST
      ASSIGN TO ....
*SORT output file
   SELECT PRODUCT-LIST
      ASSIGN TO ....
*sort file (SD)
   SELECT PRODUCT-SORT-FILE
      ASSIGN TO ....
DATA DIVISION.
FILE SECTION.
FD  ATHLETIC-SHOES-LIST.
01  A-SHOE-RECORD            PIC X(38).
FD  PRODUCT-LIST.
01  B-SHOE-RECORD            PIC X(38).
SD  PRODUCT-SORT-FILE.
01  SORT-DATA.
    05  MODEL-NAME         PIC X(10).
    05  MODEL-TYPE         PIC X(3).
    05  MODEL-NUMBER       PIC X(3).
    05  STOCK-NUMBER       PIC X(7).
    05  DESIGN-YEAR        PIC 99.
    05  UNIT-COST          PIC 999V99.
```

```
    05  UNIT-PRICE         PIC 999V99.
    05  FACTORY-NUM        PIC 999.
WORKING-STORAGE SECTION.
01   FLAGS.
     05   SHOE-LIST-EMPTY  PIC X  VALUE "N".
          88 NO-MORE-SHOE-RECORDS VALUE "Y".
     05   SORT-FILE-EMPTY  PIC X  VALUE "N".
          88 NO-MORE-SORT-RECORDS VALUE "Y".
01   HONG-KONG-NUMBER      PIC 99.
01   TAIWAN-NUMBER         PIC 99.
...
PROCEDURE DIVISION.
PRODUCT-LIST-SORT.
*temporary SD file used by sort
   SORT PRODUCT-SORT-FILE
  *major sort key
     ON ASCENDING  KEY MODEL-TYPE
*minor sort key
     ON DESCENDING KEY MODEL-NUMBER
 *duplicates sorted in the order acquired
      WITH DUPLICATES IN ORDER
         INPUT  PROCEDURE IS WEED-PRODUCT-LIST
         OUTPUT PROCEDURE IS UPDATE-PRODUCT-LIST.

WEED-PRODUCT-LIST SECTION.
OPEN-LIST-FILE.
   OPEN INPUT ATHLETIC-SHOES-LIST.
   PERFORM WEED-LIST
      UNTIL NO-MORE-SHOE-RECORDS.
   CLOSE ATHLETIC-SHOES-LIST.
   GO TO EXIT-WEED-PRODUCT-LIST.

WEED-LIST.
   READ ATHLETIC-SHOES-LIST NEXT
      AT END MOVE "Y" TO SHOE-LIST-EMPTY
      NOT AT END
*stock numbers beginning with "X" are obsolete
*do not RELEASE
        IF STOCK-NUMBER(1:1) = "X" THEN
           NEXT SENTENCE
        ELSE
*otherwise release the record to SORT
           RELEASE SORT-DATA
        END-IF.
```

```
EXIT-WEED-PRODUCT-LIST.
   EXIT.

UPDATE-PRODUCT-LIST SECTION.
CREATE-PRODUCT-LIST.
   OPEN OUTPUT PRODUCT-LIST.
   PERFORM UPDATE-RECORD
      UNTIL NO-MORE-SORT-RECORDS.
   CLOSE PRODUCT-LIST.
   GO TO EXIT-UPDATE-PRODUCT-LIST.

UPDATE-RECORD.
   RETURN PRODUCT-SORT-FILE INTO SORT-DATA
      AT END MOVE "Y" TO SORT-FILE-EMPTY
      NOT AT END
         IF FACTORY-NUM = HONG-KONG-NUMBER THEN
             MOVE TAIWAN-NUMBER TO FACTORY-NUM
         END-IF
         WRITE B-SHOE-RECORD FROM SORT-DATA.

EXIT-UPDATE-PRODUCT-LIST.
   EXIT.
```

START Statement

The Format 1 START statement modifies the current file position for a relative or indexed file. It defines a beginning point for retrieval of records from a file.

The Format 2 START statement identifies the beginning of a transaction.

General Format

Format 1

```
START file-name

[KEY IS { EQUAL TO                } key-name ]
[       { =                       }          ]
[       { GREATER THAN            }          ]
[       { >                       }          ]
[       { NOT LESS THAN           }          ]
```

```
[       { NOT <                     }          ]
[       { GREATER THAN OR EQUAL TO }          ]
[       { >=                        }          ]
[       { LESS THAN                 }          ]
[       { <                         }          ]
[       { NOT GREATER THAN          }          ]
[       { NOT >                     }          ]
[       { <=                        }          ]
[       { LESS THAN OR EQUAL TO     }          ]

[  SIZE key-size ]

[ INVALID KEY statement-1 ]

[ NOT INVALID KEY statement-2 ]

[ END-START ]
```

Format 2

```
START TRANSACTION
```

Syntax Rules

1. *File-name* is the name of a relative or indexed file with sequential or dynamic access mode.
2. For a relative file, *key-name* is the name of the RELATIVE KEY for the file. For an indexed file, *key-name* must either be the name of one of the file's record keys or the name of a data item that starts at the beginning of one of the file's record keys and is not longer than that key.
3. *Statement-1* and *statement-2* are imperative statements.
4. *Key-size* is a numeric literal or data item that specifies the maximum number of characters of the key to be examined when setting the new current file position. With this phrase, the program uses only the first "n" characters of the key to find a matching record. If "n" is zero or greater than the key size, then the entire key is used. If "n" is less than zero, the results are undefined.
5. START and TRANSACTION are required words.

General Rules

Format 1

1. The START statement changes the current file position for a file in sequential or dynamic access mode. This allows a subsequent READ NEXT or READ PREVIOUS statement to access a different record.

2. *File-name* must be open in INPUT or I-O mode when the START statement executes.

3. If the KEY phrase is missing, the implied operation is EQUAL and the implied *key-name* is the primary record key for the file (or the RELATIVE KEY for relative files).

4. The START statement changes the current file position and updates the FILE STATUS data item for *file-name*. It does not modify the record area for *file-name*.

5. The comparison specified by the KEY phrase occurs between the contents of *key-name* and the records in the file. The current file position is set according to the following rules:

   a. For EQUAL, "=", NOT LESS, NOT "<", GREATER OR EQUAL, ">=", GREATER, and ">", the current file position is set to the record with the *smallest* key that satisfies the given condition.

   b. For LESS, "<", NOT GREATER, NOT ">", LESS OR EQUAL, and "<=", the current file position is set to the record with the *largest* key that satisfies the given condition.

6. For relative files, the comparison uses the relative record numbers of the records in the file and the current value of the RELATIVE KEY data item.

7. For indexed files, the comparison uses the current value of *key-name* and the values of the corresponding data area for the records in the file.

8. If no record is found that satisfies the comparison:

   a. The invalid key condition exists.

   b. The START statement is unsuccessful.

   c. The current file position is undefined.

9. If the invalid-key condition exists, then one of the following actions occurs:

   a. If the INVALID KEY phrase is specified, *statement-1* executes.

   b. Otherwise, if an appropriate USE AFTER EXCEPTION procedure exists, it is executed.

   c. Otherwise, a message is printed and the program halts.

10. If the NOT INVALID KEY phrase is specified and the START statement is successful, *statement-2* executes.

11. If the START statement is not successful, the current file position and the current Key of Reference are undefined.

Note: The ability to START LESS, "<", NOT GREATER, NOT ">", LESS OR EQUAL, or "<=" is dependent on the host file system. Some file systems cannot process records in reverse order. These KEY phrases are all used to initiate reverse processing. If you execute one of these START statements on a machine that does not support READ PREVIOUS, you will get a file error "9B". You can test whether or not a machine has the ability to process files in this manner with the ACCEPT FROM SYSTEM-INFO verb. For details, the entry in this section for the **ACCEPT Statement**.

Format 2

1. The START TRANSACTION statement identifies the beginning of a transaction. The following rules describe how transaction management operates with Vision and relative files. For other file systems linked with the runtime, each system's native mechanism for transaction management is invoked. See the interface document for the specific file system for more details.

2. The first START TRANSACTION in a program attempts to open the log file for appending. If the log file does not exist, it is created. START TRANSACTION locks the log file, checks its integrity, then writes a START TRANSACTION notation to the log file and unlocks it.

3. After the START TRANSACTION, each Vision or relative file update operation is recorded until the next COMMIT or ROLLBACK. At that point, the record of changes is either written to the log file (COMMIT) or discarded (ROLLBACK).

4. If you compile with the "-Fs" option, an implied START TRANSACTION is in effect. Therefore, if you use this option, and then issue a START TRANSACTION, the compiler will report an error.

For more information on transaction management, see Chapter 5 of the *User's Guide.*

STOP Statement

The STOP statement terminates or suspends the program.

General Format

Format 1

```
STOP RUN [ {RETURNING} return-value ]
           {GIVING   }
```

Format 2

```
STOP literal
```

Format 3

```
STOP THREAD [ thread-id ]
```

Syntax Rules

1. *Return-value* must be a numeric literal or data item.

2. *Literal* is a numeric or alphanumeric literal.

3. *Thread-id* must be a USAGE HANDLE or HANDLE OF THREAD data item.

4. If a STOP RUN statement is in a consecutive sequence of imperative statements in a sentence, it must be the last statement in that sequence. Any statements after STOP RUN in a sentence will not execute.

5. The optional word “GIVING” or “RETURNING” may be specified before the return value in a STOP RUN statement. In this context, “GIVING” and “RETURNING” are merely commentary.

General Rules

1. A Format 1 STOP RUN statement terminates the program. Any files in the open mode are closed. This is done just as if they were the object of a CLOSE statement with no options.

2. A Format 2 STOP literal statement suspends execution of the program and passes control to the ACUCOBOL-GT debugger. The debugger prints the literal on the screen before accepting debugging commands. Note, however, that you cannot do source level debugging unless you compiled with the “-Zd” option and run with the “-d” option.

3. If *return-value* is specified, then it is assigned to the special register RETURN-CODE before the program is exited. This special register is defined as:

   ```
   77   RETURN-CODE SIGNED-LONG, EXTERNAL.
   ```

   It is implicitly shared by all programs of a run unit and is automatically created by the compiler. The final value of RETURN-CODE is returned to the host operating system when the run unit completes.

4. A Format 3 STOP THREAD statement cancels (destroys) a thread. *Thread-id* identifies the thread to cancel. If *thread-id* is omitted, the currently executing thread is canceled. If the current thread is the only thread, the STOP THREAD statement behaves like STOP RUN, except that it shuts down the runtime even if there are nested run units (see CALL RUN).

STRING Statement

The STRING statement concatenates data items.

Note: This manual entry includes code examples and highlights for first-time users following the General Rules section.

General Format

```
STRING { {source} ... [DELIMITED BY {delimiter}] } ...
                      [             {SIZE     }]

     INTO destination

   [ WITH POINTER ptr-var ]

   [ ON OVERFLOW statement-1 ]

   [ NOT ON OVERFLOW statement-2 ]

   [ END-STRING ]
```

Syntax Rules

1. *Source* and *delimiter* are nonnumeric literals or data items with USAGE DISPLAY. The “ALL literal” construct may not be used.

2. The compiler allows *source* to be a numeric literal, in which case it treats *source* as a string literal, displaying the following Warning at compilation time:

   ```
   Warning: Literal is numeric - treated as alphanumeric
   ```

   In such cases, leading zeros are stripped from the numeric literal to form the string literal.

3. *Destination* is a data item with USAGE DISPLAY. It may not be JUSTIFIED or edited, but may be reference modified.

4. *Ptr-var* is an integer numeric data item.

5. *Statement-1* and *statement-2* are imperative statements.

6. The size of *ptr-var* must allow it to contain a value one greater than the size of *destination*.

7. The DELIMITED phrase may be omitted only immediately before the INTO phrase. If omitted, DELIMITED BY SIZE is implied.

General Rules

1. The STRING statement concatenates the *source* values and places the result in *destination*. There are no limitations (other than available memory) on the number of *source* items allowed.

2. The STRING statement moves characters from *source* to *destination* according to the rules for alphanumeric to alphanumeric moves. However, no space filling occurs.

3. The contents of each *source* item are moved to *destination* in the order they appear. Data is moved from the left, character by character, until the end of *source* is reached. The end point of each *source* item is determined by the DELIMITED phrase according to the following rules:

   a. Transfer stops when the end of *source* is reached.

   b. Transfer stops when the end of *destination* is reached.

   c. If *delimiter* is specified, transfer stops when the characters specified by *delimiter* are found in *source*. The *delimiter* is not included in the characters transferred.

   d. If the SIZE option is used, transfer ends only when the end of *source* or *destination* is reached.

   e. When *source* or *delimiter* is a figurative constant, it represents a size of one character.

   f. If *source* is a variable size item, the current size is used to determine the end of *source*. If the current size is zero characters, no transfer occurs for that *source* item.

4. When the POINTER phrase is specified, *ptr-var* must be set by the program to a value greater than zero before the STRING statement executes. The transfer to *destination* starts at the character position in *destination* indicated by *ptr-var*. The leftmost character of *destination* is position “1”. If no POINTER phrase is used, an implied pointer is created and set to the value “1”.

5. *Ptr-var* (or the implied pointer) is incremented by one for each character transferred to *destination*. The transfer to *destination* always occurs at the character position indicated by the current pointer value.

6. When the STRING statement ends, only those parts of *destination* referenced during execution change.
7. Before moving each character to *destination*, the STRING statement tests the value of *ptr-var* (or the implied pointer if *ptr-var* is not specified). If this value is less than one or greater than the size of *destination*, the overflow condition is set and the following happens:
   a. No more data is transferred to destination.
   b. If the ON OVERFLOW phrase is used, *statement-1* executes.
   c. The STRING statement ends.
8. If the NOT ON OVERFLOW phrase is specified, *statement-2* executes after the STRING statement is finished if the overflow condition has not occurred (see general rule 7).
9. Subscripting for *source* and *delimiter* occurs just before the corresponding item is used.
10. Subscripting for *ptr-var* and *destination* occurs just before the STRING statement executes.

Code examples

Assume the following data items:

```
01  CLAIM-CODE   PIC X(20).  |destination data item,
                             |initialize before use
01  CUSTOMER-ID  PIC X(8).   |source data item
01  ORDER-NO     PIC X(10).  |source data item
01  ORDER-DATE   PIC 9(6).   |source data item
01  STRING-PTR   PIC 99.     |concatenation pointer,
                             |initialize before use
```

Assume the program assigns the following values before the STRING statement executes:

```
CLAIM-CODE  = SPACES
(destination item gets space filled)
CUSTOMER-ID = C077/W12
(customer number "/" region code)
ORDER-NO    = W12-A00234
(region code "-" order number)
ORDER-DATE  = 060199
```

```
(mmddyy)
```

Example 1:

```
STRING ORDER-DATE DELIMITED BY SIZE
       ORDER-NO   DELIMITED BY SIZE
       INTO CLAIM-CODE
END-STRING.
*CLAIM-CODE = "060199W12-A00234    "
*spaces appear at the end of concatenated string
*because CLAIM-CODE was assigned SPACES before the
*STRING concatenation statement executed
```

Example 2:

Use POINTER to coordinate multiple STRING statements into a common concatenation object.

```
SET STRING-PTR TO 1.
MOVE SPACES TO CLAIM-CODE.
STRING ORDER-DATE DELIMITED BY SIZE
    INTO CLAIM-CODE
    POINTER STRING-PTR
END-STRING.
*CLAIM-CODE now contains: "060199"
*Reassign the value of STRING-PTR so as to
*eliminate the year digits by overwriting
*positions 5 & 6 in the destination string
SUBTRACT 2 FROM STRING-PTR.
...
*build the remainder of the string
*start by adding a hyphen delimiter,
*add all characters before "/" in CUSTOMER-ID
*add another hyphen delimiter
*add all of ORDER-NO
STRING "-",
       CUSTOMER-ID DELIMITED BY "/",
       "-",
       ORDER-NO    DELIMITED BY SIZE
   INTO CLAIM-CODE
   POINTER STRING-PTR
   ON OVERFLOW
      DISPLAY "Claim-Code OVERFLOW"
   NOT ON OVERFLOW
      PERFORM PROCESS-CLAIM-CODE
```

```
END-STRING.
*CLAIM-CODE = "0601-C077-W12-A00234"
```

Highlights for first-time users

1. A clear, concise description of the concatenation transfer process is contained in entry 3 of the preceding General Rules section.
2. Use DELIMITED BY to concatenate a portion of the source item up to, but not including, the delimiter. The delimiter may be a single character or a string.
3. Use OVERFLOW to do special processing in the event that the size of the concatenation overflows the destination data item. The OVERFLOW phrase should always be included when an overflow condition is possible.
4. Use NOT ON OVERFLOW to do special processing in the event that the concatenation succeeds (does not result in an overflow).
5. Use POINTER to place data into a common destination when concatenation requires multiple STRING statements. See code example 2, above.
6. The STRING statement does not space fill the target data item. You must initialize the destination data item. For example:

   ```
   01 CLAIM-CODE  PIC X(20)  VALUE ALL SPACES.
   ```

SUBTRACT Statement

The SUBTRACT statement performs arithmetic subtraction.

General Format

Format 1

```
SUBTRACT {number} ... FROM { result [ROUNDED] } ...

   [ ON SIZE ERROR statement-1 ]

   [ NOT ON SIZE ERROR statement-2 ]
```

```
   [ END-SUBTRACT ]
```

Format 2

```
SUBTRACT {number} ... FROM number

     GIVING { result [ROUNDED] } ...

   [ ON SIZE ERROR statement-1 ]

   [ NOT ON SIZE ERROR statement-2 ]

   [ END-SUBTRACT ]
```

Format 3

```
SUBTRACT {CORRESPONDING} group-1 FROM group-2 [ROUNDED]
         {CORR         }

   [ ON SIZE ERROR statement-1 ]

   [ NOT ON SIZE ERROR statement-2 ]

   [ END-SUBTRACT ]
```

Format 4

```
SUBTRACT TABLE src-table FROM dest-table [ROUNDED]

   [ FROM INDEX src-start TO src-end ]

   [ DESTINATION INDEX dest-start ]

   [ ON SIZE ERROR statement-1 ]

   [ NOT ON SIZE ERROR statement-2 ]

   [ END-SUBTRACT ]
```

Syntax Rules

1. *Number* is a numeric literal or elementary numeric data item.
2. *Result* is an elementary numeric data item or, in Format 2, an elementary numeric edited data item.

3. *Group-1* and *group-2* are group items containing one or more elementary numeric data items.

4. *Statement-1* and *statement-2* are imperative statements.

5. CORR is an abbreviation of CORRESPONDING.

6. *Src-table* and *dest-table* are numeric data items that are table elements. The low-order subscript of these items must be omitted. For example, if "SRC-1" was an element of a one-dimensional table, you would use "SRC-1" in the statement. If "SRC-2" was an element of a two-dimensional table, and you wanted to add all the elements in row "2", you would use "SRC-2(2)".

7. *Src-start*, *src-end* and *dest-start* are numeric literals or data items. These items may not be subscripted.

General Rules

1. In Format 1, the *numbers* are added together and the result is then subtracted from each *result* in turn.

2. In Format 2, the *numbers* before the word FROM are added together and the result is subtracted from the *number* following the word FROM. The results are then moved to each *result* item.

3. In Format 3, elementary numeric items in *group-1* are subtracted from the corresponding items in *group-2*. The values are then stored in *group-2*.

4. Additional information can be found in the sections covering **Arithmetic Operations** (6.4.1), **Multiple Receiving Fields** (6.4.2), the **ROUNDED Option** (6.4.3), the **SIZE ERROR Option** (6.4.4), and the **CORRESPONDING Option** (6.4.5).

5. A Format 4 SUBTRACT statement subtracts a range of *src-table* elements from a range of *dest-table* elements. The results are stored in *dest-table*. The first element of the *src-table* range is subtracted from the first element of the *dest-table* range, the second element from the second, and so on.

6. *Src-start* indicates the first element of the source range. If omitted, it defaults to "1". *Src-end* indicates the last element of the range (inclusive). If omitted, it is set to the current upper bound of the source table. In a multidimensional table, the range of elements varies over the innermost OCCURS.

7. *Dest-start* indicates the first element of the destination range. If omitted, it defaults to "1". Note that the last element of the destination range is *dest-start* + *src-end* - 1.

8. If the SIZE ERROR phrase is used, elements for which the size error condition exists are not updated, while the remaining elements are. If any element gets a size error, then *statement-1* executes, otherwise *statement-2* executes.

Note: SUBTRACT TABLE is usually substantially faster than the equivalent PERFORM loop. The degree of improvement depends on the size of the range (larger ranges show better improvement). The runtime always performs table boundary checking in SUBTRACT TABLE, even if you do not compile with "-Za". The boundaries are tested only at the end points (this is a fast test and there is nothing to be gained from not performing it).

UNLOCK Statement

The UNLOCK statement removes file record locks.

General Format

Format 1

```
UNLOCK file-name ALL {RECORD }
                     {RECORDS}
```

Format 2

```
UNLOCK ALL RECORDS
```

Format 3

```
UNLOCK THREAD
```

Syntax Rule

File-name is a relative or indexed file described in the Data Division.

General Rules

Format 1

1. *File-name* must be in the open mode when the UNLOCK statement executes.
2. When the UNLOCK statement executes, the currently locked records (if any) for *file-name* are unlocked and made available to other users. The statement has no effect if there are no records locked for *file-name*.
3. The UNLOCK statement will fail only when the file is not open. The UNLOCK statement updates the file's associated FILE STATUS data item.
4. During a transaction, the UNLOCK statement affects only those files for which rollback has *not* been enabled. In the case where the UNLOCK statement is ineffective because rollback has been enabled for the file, the file status will be set to 00 (success).

Format 2

1. A Format 2 UNLOCK statement releases all records locked by all of the open files in the program.
2. A Format 2 UNLOCK always succeeds. No file status data items are updated by this verb and no Declarative procedures are ever executed.

Format 3

A Format 3 UNLOCK THREAD statement removes the last lock applied to the thread. If the thread has only one lock (only one LOCK THREAD statement has executed in the thread), then the UNLOCK THREAD statement has the effect of allowing other threads to run. If more than one lock has been applied to the thread, then the UNLOCK THREAD statement removes the last lock applied to the thread and the thread remains locked until it has been unlocked as many time as it was locked. This allows a thread to

lock itself, call a subroutine that also locks itself, and remain locked when that subroutine unlocks itself. See the entry in this section for LOCK THREAD.

If the current thread is not locked, the UNLOCK THREAD statement has no effect.

UNSTRING Statement

The UNSTRING statement separates a data item into one or more receiving fields. Delimiters may be used to specify the ends of fields. Substring values are assigned to unique destination data items.

Note: This manual entry includes code examples and highlights for first-time users following the General Rules section.

General Format

```
UNSTRING source

   [ DELIMITED BY [ALL] delim

                 [ OR [ALL] delim ] ... ]

     INTO { dest [ DELIMITER in delim-dest ]

                [ COUNT IN counter ] } ...

   [ WITH POINTER ptr-var ]

   [ TALLYING IN tally-var ]

   [ ON OVERFLOW statement-1 ]

   [ NOT ON OVERFLOW statement-2 ]

   [ END-UNSTRING ]
```

Syntax Rules

1. *Source* is an alphanumeric data item. *Source* may be reference modified.
2. *Dest* is a USAGE DISPLAY data item. It may not be edited.
3. *Delim* is a nonnumeric literal or an alphanumeric data item. The "ALL literal" construct may not be used.
4. The compiler allows *source* and *delim* to be numeric literals, in which case it treats them as string literals, displaying the following Warning at compile time:

   ```
   Warning: Literal is numeric - treated as alphanumeric
   ```

   In such cases, leading zeros are stripped from the numeric literal to form the string literal.
5. *Delim-dest* is an alphanumeric data item.
6. *Counter*, *ptr-var*, and *tally-var* are integer numeric data items.
7. *Statement-1* and *statement-2* are imperative statements.
8. *Ptr-var* must be large enough to contain a value one greater than the size of *source*.
9. The DELIMITER IN and COUNT IN phrases can appear only if there is a DELIMITED BY phrase.

General Rules

1. UNSTRING breaks up *source* into the various *dest* fields. *Source* is the sending field and *dest* is the receiving field. Up to 50 *dest* items are allowed.
2. *Counter* represents the count of the number of characters within *source* isolated by the delimiters for the move to *dest*. This does not include a count of the delimiter characters.
3. *Ptr-var* represents the relative character position within *source* to move from. The leftmost position is position "1". If no POINTER phrase is specified, examination begins with the leftmost character position.

4. *Tally-var* is a counter which is incremented by 1 for each *dest* item accessed during the UNSTRING operation.

5. Neither *ptr-var* nor *tally-var* is initialized by the UNSTRING statement.

6. Each *delim* represents one delimiter. When a delimiter contains two or more characters, all the characters must be present in contiguous positions in *source* to be recognized as a delimiter. When *delim* is a figurative constant, it stands for a single nonnumeric literal.

7. When the ALL phrase is specified, one or more contiguous occurrences of *delim* in *source* are treated as if they were only one occurrence for the remaining General Rules. Only one occurrence of *delim* is moved to *delim-dest* in this case.

8. When two or more delimiters are specified, an OR condition exists between them. Each delimiter is compared to the sending field in the order written. If a match occurs, the characters in the sending field are considered to be a single delimiter. No characters in *source* can be considered a part of more than one delimiter.

9. When an examination encounters two contiguous delimiters, the current receiving area is space-filled if it is alphabetic or alphanumeric, or zero-filled if it is numeric.

10. When the UNSTRING statement initiates, the current receiving area is the first *dest* item. Data is transferred from *source* to the receiving area according to the following rules:

    a. Examination starts at the character position indicated by *ptr-var*, or the leftmost position if *ptr-var* is not specified.

    b. If the DELIMITED BY phrase is specified, the examination proceeds left-to-right until a delimiter is encountered. If the DELIMITED BY phrase is not specified, the number of characters examined is equal to the size of the receiving area. The sign character of the receiving item (if any) is not included in the size. If the end of *source* is encountered before the delimiting condition is met, the examination stops with the last character of *source*.

c. The characters examined (excluding the delimiting characters, if any) are treated as an elementary alphanumeric item. These characters are moved to the current receiving field according to the rules for the MOVE statement, including space filling.

d. If the DELIMITER IN phrase is specified, the delimiting characters are moved to *delim-dest* as if they were the alphanumeric source of a MOVE statement. If the delimiting condition is the end of *source*, then *delim-dest* is space-filled.

e. If the COUNT IN phrase is specified, the number of characters examined (excluding the delimiter) is moved to *counter* as if the count were the numeric source of a MOVE statement.

f. If the DELIMITED BY phrase is specified, the *source* item is further examined beginning with the first character to the right of the delimiter found. If the DELIMITED BY phrase is not specified, the *source* item is further examined beginning with the character to the right of the last character examined.

g. The current receiving area is then set to the next *dest* item and the cycle specified in steps (b) through (g) is repeated until either all the characters in *source* are examined or there are no more *dest* items.

11. The *ptr-var* (if any) is incremented by 1 for each character in *source* examined.

12. An overflow condition occurs in either of the following situations:

    a. The value of *ptr-var* is less than one or greater than the size of *source* when the UNSTRING statement starts.

    b. During execution, all *dest* items have been acted upon and *source* contains unexamined characters.

13. When the overflow condition exists, *statement-1* (if any) executes and the UNSTRING statement terminates.

14. If *statement-2* is specified, it executes after the UNSTRING statement has finished if the overflow condition has not occurred.

Code examples

Use UNSTRING to decompose strings containing multiple data elements. For example, a string data item might contain a person's name, using commas to separate the name fields: "last-name,first-name,middle-initial". Using UNSTRING, and specifying "," (comma) as the delimiter, you could separate the name string into three data items, each containing an element of the full name.

Example 1:

Assume the following data items:

```
01  CUSTOMER-NAME    PIC X(40)  VALUE ALL SPACES.
01  LAST-NAME        PIC X(25)  VALUE ALL SPACES.
01  FIRST-NAME       PIC X(14)  VALUE ALL SPACES.
01  MIDDLE-I         PIC X      VALUE ALL SPACES.
{ . . . }
PROCEDURE DIVISION.
{ . . . }
 DISPLAY 'Enter name: LAST,FIRST,MIDDLE-INITIAL'.
 DISPLAY 'Use a comma to separate each name entry'.
    ACCEPT CUSTOMER-NAME.

{ . . . }

UNSTRING CUSTOMER-NAME
   DELIMITED BY ","
   INTO LAST-NAME,   |characters to first comma
        FIRST-NAME,  |characters to second comma
        MIDDLE-I     |gets only the first character
                     |of the remaining string.  No
                     |overflow is raised.
                     |See general rule 12.
   ON OVERFLOW
      DISPLAY 'OVERFLOW on UNSTRING'
END-UNSTRING.
```

For code examples 2 and 3 assume the following data items:

```
01  COLOR-LIST  PIC X(22) VALUE "RED:BLUE/GREEN  YELLOW".
01  COLOR-1     PIC X(6)  VALUE ALL SPACES.
01  COLOR-2     PIC X(6)  VALUE ALL SPACES.
01  COLOR-3     PIC X(6)  VALUE ALL SPACES.
```

```
01  COLOR-4     PIC X(6)  VALUE ALL SPACES.
01  DELIMIT-1   PIC X(3)  VALUE ALL SPACES.
01  COUNT-1     PIC 9     VALUE 0.
```

Example 2:

```
UNSTRING COLOR-LIST
   DELIMITED BY ":" OR "/" OR ALL SPACE
*ALL SPACE treats contiguous spaces
*as one delimiter.
   INTO COLOR-1,
        COLOR-2,
        COLOR-3,
        COLOR-4
END-UNSTRING.
*COLOR-1 = "RED   "
*COLOR-2 = "BLUE  "
*COLOR-3 = "GREEN "
*COLOR-4 = "YELLOW"
```

Example 3:

```
MOVE 0 TO COUNT-1.

UNSTRING COLOR-LIST
   DELIMITED BY ":" OR "/" OR ALL SPACE
*DELIMIT-1 and COUNT-1 will hold only
*the values associated with COLOR-1.
   INTO COLOR-1
         DELIMITER IN DELIMIT-1
         COUNT IN COUNT-1,
         COLOR-2,
         COLOR-3,
         COLOR-4
   ON OVERFLOW
      DISPLAY "overflow: unstring colors"
   NOT ON OVERFLOW
*do when UNSTRING succeeds.
      PERFORM SORT-COLORS
END-UNSTRING.
*COLOR-1 = "RED   "
*COLOR-2 = "BLUE  "
*COLOR-3 = "GREEN "
*COLOR-4 = "YELLOW"
*DELIMIT-1 = ":  "
```

```
*COUNT-1 = 3 count-1 holds the number of characters in RED
```

Example 4:

When the string does not contain delimiters between the data elements, but the size and position of each string data element is known, the string can be deconstructed without a DELIMITED BY phrase.

Assume the following data items:

```
01  COLOR-LIST   PIC X(7) VALUE "REDBLUE".
01  COLOR-1      PIC X(3) VALUE ALL SPACES.
01  COLOR-2      PIC X(4) VALUE ALL SPACES.
{ . . . }
PROCEDURE DIVISION.
{ . . . }
UNSTRING COLOR-LIST
   INTO COLOR-1,
*first substring must be three characters.
        COLOR-2
*second substring must be four characters.
END-UNSTRING.
*COLOR-1 = "RED"
*COLOR-2 = "BLUE"
```

Example 5:

Use POINTER and a PERFORM loop to extract and process string elements.

Assume the following data items:

```
01  COLOR-LIST       PIC X(21)  VALUE "RED BLUE GREEN YELLOW".
01  COLOR-LIST-SIZE  PIC 999.
01  COLOR-1          PIC X(6)   VALUE SPACES.
01  STRING-PTR       PIC 99.
01  FLAGS.
    05  COLOR-STRING-EMPTY   PIC X VALUE "N".
        88 NO-MORE-COLORS          VALUE "Y".
{ . . . }
PROCEDURE DIVISION.
{ . . . }
*string pointer must be initialized
MOVE 1 TO STRING-PTR.
SET COLOR-LIST-SIZE TO SIZE OF COLOR-LIST.
```

```
PERFORM PROCESS-COLOR UNTIL NO-MORE-COLORS.
{ . . . }
PROCESS-COLOR.
   UNSTRING COLOR-LIST
      DELIMITED BY ALL SPACE
      INTO COLOR-1
      POINTER STRING-PTR
      ON OVERFLOW
*An OVERFLOW condition will be raised every time
*through the loop, except when extracting the last
*substring.  When the overflow is the result of
*having unexamined characters at the end of the
*input string, take no action.  When the overflow
*is due to the pointer value exceeding the length
*of the string, set COLOR-STRING-EMPTY.
         IF STRING-PTR > COLOR-LIST-SIZE THEN
            MOVE "Y" TO COLOR-STRING-EMPTY
         END-IF
*process the value
   PERFORM STORE-COLOR-1
*initialize COLOR1 before fetching the next color
   MOVE SPACES INTO COLOR-1
   END-UNSTRING.
```

Highlights for first-time users

1. UNSTRING is best suited for separating string components that share a common delimiter. The delimiter must not appear as an element of the components' values.

2. DELIMITED BY is optional. If it's omitted, each destination data item is completely filled. Effectively, the respective size of each destination data item is the respective delimiter.

3. Assignment to the destination data item is done with an implied MOVE. The MOVE operation will truncate the substring or space fill the destination data item, as required. Truncation of the substring, or space filling of the destination data item resulting from the implicit MOVE, does not raise an OVERFLOW condition.

4. The OVERFLOW condition is raised if: (a) all destination data items are used and characters still remain in the source data item; or (b) POINTER is used and the value of the pointer variable is less than 1 or greater than the length of the source data item.

5. Use the ALL option to treat contiguous occurrences of a delimiter, such as spaces, as a single occurrence.

6. Use DELIMITER IN to place the delimiting character(s) of the current substring into the named data item.

7. Use the COUNT IN option to save the length of the current substring into the named data item.

8. Use TALLYING to tally the number of destination data items assigned by the UNSTRING statement.

9. Use the POINTER option to specify a numeric holder *(ptr-var)* for the current position in the source data item. By pre-assigning a value to the pointer variable you can start the examination of the source data item at any position in the string. *Ptr-var* is incremented by one for each character in the source data item that is examined. POINTER allows the programmer to use multiple UNSTRING statements to process the source data item. Note, however, that an overflow condition will be raised if the value of *ptr-var* is less than the length of the string when the UNSTRING statement terminates.

   You must initialize the tallying and pointer variables or results are unpredictable.

10. Use the OVERFLOW option to do special processing when the UNSTRING process does not examine every character in the source data item, or when the pointer variable has a value of less than one or more than the length of the source data item. When the overflow condition exists, the associated imperative statement (if any) executes and program execution continues immediately after the UNSTRING statement.

11. Use the NOT ON OVERFLOW option to do special processing when the UNSTRING statement processes the entire source data item.

USE Statement

The USE statement specifies procedures for handling Input/Output errors and other errors. USE is a comprehensive error handling construct. The USE statement locates all error routines centrally within the DECLARATIVES

section of the PROCEDURE DIVISION when used to specify I/O error handling routines. USE is a valuable supplement to the AT END and INVALID KEY I/O error handling phrases.

Note: This manual entry includes code examples and highlights for first-time users following the General Rules section.

General Format

Format 1

```
USE AFTER STANDARD {EXCEPTION} PROCEDURE ON { {file}...   }
                   {ERROR    }             { INPUT       }
                                           { OUTPUT      }
                                           { I-O         }
                                           { EXTEND      }
                                           { TRANSACTION }
```

Format 2

```
USE AFTER STANDARD {EXCEPTION} PROCEDURE ON OBJECT
                   {ERROR    }
```

Format 3

```
USE FOR REPORTING ON {index-file} ...
```

Format 4

```
USE AFTER STANDARD ERROR PROCEDURE ON file-name GIVING
       data-name-1  [ data-name-2 ]
```

Format 5

```
USE { active-x-control-item }
    { com-object-item  }
    { property-1 [ ( param-1 ... ) ]
      [ :: property-2 [ ( param2 ... ) ] ] ... }
    { statement }
[END-USE]
```

Format 6

```
USE AT PROGRAM {START}
               {END  }
```

Syntax Rules

1. *File* is a file described in the Data Division. It may be a sort file.
2. *Index-file* is a file described in the Data Division. It must be an indexed file.
3. *Data-name-1* is an eight-byte data item of the type PICTURE 9(8) USAGE DISPLAY. A compile-time error message is generated if *data-name-1* is not of that type.
4. *Data-name-2* may be of any type but must be at least as long as the file buffer. A compile-time error message is generated if this is not the case.
5. When used, a USE statement must immediately follow a section header in the Declaratives portion of the Procedure Division and must appear in a sentence by itself. The remainder of the section must consist of zero, one, or more paragraphs that define the exception procedure to be used.
6. ERROR and EXCEPTION are interchangeable.
7. The INPUT, OUTPUT, I-O, and EXTEND and TRANSACTION phrases may each be specified in only one USE statement in a Procedure Division.
8. A particular *file* may not appear in more than one Format 1 USE statement in a program.
9. A particular *index-file* may not appear in more than one Format 3 USE statement in a program.
10. *Active-x-control-item* and *com-object-item* must be USAGE HANDLE OF *class-name*, where *class-name* is defined as an ActiveX control or an COM object in SPECIAL-NAMES.
11. *Statement* is an imperative statement.
12. *Property-1* is the name of a property of the ActiveX control or COM object. *Property-1* must not be a write-only property.
13. *Property-2* is the name of a property of the ActiveX control or COM object which is the value of *property-1*. *Property-2* must not be a write-only property.

14. *Param-1* and *param-2* are literals, data items, or numeric expressions.

General Rules

1. The USE statement is not executed; it merely defines the conditions calling for the execution of the USE procedure. The USE procedure consists of all the paragraphs contained in the section the USE statement appears in.

2. The procedure associated with a Format 1 USE statement is executed after the unsuccessful execution of an I/O operation unless an AT END or INVALID KEY phrase takes precedence. It also executes *during* an I/O operation when a duplicate key error is detected for a file open for BULK-ADDITION. The following rules apply:

   a. If *file* is specified, the associated procedure is executed when an unsuccessful I/O operation occurs for that file.

   b. If the INPUT phrase is specified, the procedure executes when the OPEN operation is unsuccessful for a file being opened in INPUT mode, unless that file is specified by *file* in another USE statement. This also applies to a file being opened for INPUT.

   c. The OUTPUT, I-O, and EXTEND phrases operate as described in rule b), except that they apply to files opened in the corresponding mode.

   d. If TRANSACTION is specified, the procedure executes when an error occurs during a START TRANSACTION, COMMIT, ROLLBACK, or call to C$RECOVER. Note that the status-code will be in the TRANSACTION-STATUS variable. See Appendix E, Book 4, *Appendices* for a list of transaction status codes.

3. After the USE procedure executes, control is returned to the next executable statement after the I/O statement that caused the USE procedure to execute. If the USE procedure executed during a file operation, the control returns to that file operation instead.

4. Within a USE procedure, no statement may be executed that would result in the execution of a USE procedure that has been invoked but has not yet returned.

5. The procedure associated with a Format 2 USE statement is executed after an object exception occurs.

6. After the Format 2 USE statement executes, control is returned to the next executable statement after the statement that caused the USE procedure to execute.

7. A Format 2 USE statement executes when an object exception is "raised" during a DISPLAY, MODIFY, INQUIRE or calls to C$GETEVENTDATA, C$SETEVENTDATA, C$GETEVENTPARAM or C$SETEVENTPARAM.

   An object exception can either be raised by the object itself or by the runtime to indicate that an error has occurred. ActiveX controls and COM objects are currently the only objects that can raise exceptions. These are called COM exceptions in Microsoft terminology.

   Information about an object exception can be retrieved with the C$EXCEPINFO routine.

8. Within a Format 2 USE statement, no statement can be executed that would result in the execution of a USE procedure that has been invoked but has not yet returned.

9. A Format 3 USE statement executes at periodic times for files opened with the BULK-ADDITION phrase. The purpose of this procedure is to report to the user the progress of writing keys for a large number of records written to *index-file*. See section 6.1.6.3 of the *ACUCOBOL-GT User's Guide* for more details.

10. When a Format 1 USE procedure executes due to a duplicate key error for a file open with BULK-ADDITION, no file I/O statements may be executed. This also applies to Format 3 USE procedures. In addition, no run units may be started or stopped, and the program containing the declarative may not perform an EXIT PROGRAM.

11. A Format 4 USE procedure is valid only when the "-Cv" option is in effect. When an error handler introduced by this statement is invoked, the runtime puts special error codes into the eight-byte data item *data-item-1*. See Chapter 5, "IBM DOS/VS COBOL Conversions," in *Transitioning to ACUCOBOL-GT* for complete information on DOS/VS COBOL compatibility mode. See Appendix E.5 for the IBM DOS/VS COBOL error codes.

12. If *data-name-2* is present, when the error handler is invoked, it will also load *data-name-2* with the contents of the file buffer. ACUCOBOL-GT always loads *data-name-2*, and if the data-item is larger than the file buffer, the excess bytes at the right end are left unchanged.

13. The Format 5 USE verb sets up a context for more efficient coding and processing of MODIFY and INQUIRE statements that operate on ActiveX controls or objects. It allows you to execute a series of MODIFY and INQUIRE statements on a specified object without respecifying the object. For example, to change a number of different properties on a single object, place the MODIFY statement within the USE statement, referring to the object once instead of referring to it in each MODIFY clause.

```
USE MyChart Legend::Font
MODIFY    ^Size = 10
          ^Name = "Courier"
          ^Bold = 1
END-USE
```

14. *Param-1* is the first parameter passed when getting the value of *property-1*.

15. *Param-2* is the first parameter passed when getting the value of *property-2*.

16. Runtime errors announced as "Use of a LINKAGE data item not passed by the caller" and "Passed USING item smaller than corresponding LINKAGE item" belong to the class of "intermediate" runtime errors that, upon occurrence, call installed error procedures.

17. When placed in a program's Declarative section, a Format 6 USE statement creates a START or END procedure for the program. Each program may contain no more than one START and one END procedure. Every program in a run unit may contain such procedures.

    a. A START procedure executes immediately before the first normal COBOL statement in the Procedure Division when the program is in its initial state. The START procedure executes only once regardless of the number of times the program is entered, until the program is returned to its initial state (e.g. via CANCEL). A

START procedure executes regardless of which entry point is used to start the program when a program contains multiple entry points (see the ENTRY statement).

b. An END procedure executes immediately before a program is placed into its initial state or it is about to leave memory, providing the program has been entered at least once. An END procedure executes before open files are closed as part of the shutdown process.

c. You can call the C$EXITINFO library routine from an END procedure to obtain information about the program exit. For example, you can determine if the exit is the result of a STOP RUN or a fatal error. Please refer to Appendix I in *ACUCOBOL-GT Appendices* for detailed information about this library routine.

d. It is normal for an END procedure to execute when the program that contains it is inactive. For example, if a program is canceled, its END procedure will execute when the program is otherwise inactive. For this reason, an END procedure should not reference data passed to the program through Linkage. This data will not be defined in many cases.

e. END procedures are executed during abnormal shutdown when possible. However, certain operating system errors (such as a fatal memory error) cannot be caught in some operating environments, and in these cases the END procedures will not be able to execute. If a fatal error occurs during an END procedure, that procedure stops, but other unprocessed END procedures execute where possible.

f. When multiple END procedures execute (e.g. STOP RUN when several programs are in memory), the order of their execution is arbitrary.

Code example

Format 1:

```
PROCEDURE DIVISION.
DECLARATIVES.
NAMED-FILE-IO-ERROR-HANDLING SECTION.
```

```
   USE AFTER STANDARD ERROR PROCEDURE ON
      REPORT-FILE.
NAMED-FILE-IO-ERROR-HANDLER.
{ . . . }
IO-INPUT-ERROR-HANDLING SECTION.
   USE AFTER STANDARD ERROR PROCEDURE ON INPUT.
IO-INPUT-ERROR-HANDLER.
{ . . . }
IO-OUTPUT-ERROR-HANDLING SECTION.
   USE AFTER STANDARD ERROR PROCEDURE ON OUTPUT.
IO-OUTPUT-ERROR-HANDLER.
{ . . . }
END DECLARATIVES.
MAIN-PROGRAM SECTION.
{ . . . }
```

Highlights for first-time users

1. The USE statement can be used to handle program file I/O errors. USE is not executed, but, rather, describes the conditions under which the contained procedures are to be executed.

2. The USE statement is located in a DECLARATIVES section in a program's PROCEDURE DIVISION. The USE statement may contain one or many error handling procedures. Each USE statement may specify a file, set of files or OPEN mode for which the enclosed procedures apply. No file name or OPEN mode (INPUT, OUTPUT, I-O, EXTEND) may be named more than once in the DECLARATIVES section of that PROCEDURE DIVISION. If an I/O error raises an ambiguity between an error handling procedure that names the file and an error handling procedure with a matching OPEN mode description, the procedure naming the file takes precedence.

3. Detecting and handling I/O errors:

   Every I/O operation returns a two-digit status code that indicates the result of the operation. A status code that begins with "0" indicates a successful operation. A status code that begins with a number other than "0" indicates that the I/O operation failed. For a complete list of file status codes see Appendix E, Book 4, *Appendices.*

   When an I/O operation takes place, the file status code is set and sent back to the calling statement.

- If the status code begins with the number “1”, an AT END error has occurred.
- If the status code begins with a “2”, an INVALID KEY error has occurred.
- If the programmer has included a corresponding AT END or INVALID KEY phrase in the I/O statement (where supported), the respective clause is executed and the program, if not terminated, continues after the I/O statement that raised the error.
- If the status code begins with any other number (except “0”), or there is no AT END or INVALID KEY phrase, the system searches for an applicable USE statement in the DECLARATIVES section.
- If an applicable USE statement is found, the search stops, the applicable error handler is executed, and if the program has not been terminated, program execution continues after the I/O statement that raised the error.
- If no applicable USE statement is found, the runtime determines the action. Usually, a message is presented and the program halts.

For more about file status and the AT END, and INVALID KEY phrases, see **section 6.4.8**.

4. The set of I/O verbs that return a status code includes: CLOSE, DELETE, OPEN, READ, REWRITE, START, UNLOCK, WRITE.

5. There are five different standards specifying the values of file status codes: ANSI85, ANSI74, DG ICOBOL, VAX COBOL, and IBM DOS/VS COBOL. See Appendix E, Book 4, for the complete definitions of the status codes corresponding to each standard. By default, ACUCOBOL-GT uses the ANSI85 status code standard. You can change to any of the alternate standards by changing the setting of the “**FILE_STATUS_CODES**“ runtime configuration variable.

   The ANSI85 (default) definitions of the major error classes are:

Status Code	**Status**
0*x*	I/O operation succeeded

Status Code	Status
1*x*	AT END ERROR
2*x*	INVALID KEY ERROR
3*x*	PERMANENT ERROR
4*x*	LOGIC ERROR
9*x*	ACUCOBOL-GT DEFINED

Note: Some errors such as 30, 98, and 9D also return additional information in the secondary or tertiary file codes. These may be retrieved with the library routine C$RERR.

WAIT Statement

The WAIT statement synchronizes operations between threads.

General Format

```
WAIT FOR { THREAD thread-ID          }
         { LAST THREAD               }
         { ANY THREAD                }
```

Remaining phrases are optional, can appear in any order.

```
 { BEFORE TIME timeout }
 { TEST ONLY           }

 THREAD IN thread-2

 SIZE IN size-item

 STATUS IN status-item
 [ ON EXCEPTION statement-1 ]

 [ NOT ON EXCEPTION statement-2 ]

 [ END-WAIT ]
```

Syntax Rules

1. *Thread-ID* and *thread-2* are usage HANDLE or HANDLE OF THREAD data items. *Thread-2* may not be indexed or reference modified.
2. *Timeout* is a numeric literal or data item.
3. *Size-item* is a numeric data item. It may not be indexed or reference modified.
4. *Status-item* is a two-character group item, PIC XX, or PIC 99 data item. It may not be indexed or reference modified.
5. *Statement-1* and *statement-2* are any imperative statements.

General Rules

1. The WAIT statement waits for a thread to terminate or send a message. The thread waited on is determined as follows:
   a. FOR THREAD *thread-ID* specifies the thread identified by *thread-ID*.
   b. FOR LAST THREAD specifies the *last* thread (see section 6.8.1, Book 1, *ACUCOBOL-GT User's Guide* for a discussion of the *last* thread).
   c. FOR ANY THREAD specifies all threads. The first one to terminate or send a message satisfies the WAIT statement
2. If a message is available when the WAIT statement executes, then WAIT statement finishes immediately.
3. When BEFORE TIME is specified, the WAIT statement will time-out after the specified *(timeout)* number of hundredths of seconds. If the WAIT statement times out before receiving a message, it terminates with an exception condition and it does not modify *thread-2* or *size-item*. If *timeout* is zero, then the WAIT statement times-out immediately if a message is not available. Specifying TEST ONLY is equivalent to specifying a *timeout* value of zero.

4. If the thread waited for does not exist or terminates before sending a message, then the WAIT statement terminates with an exception condition and does not modify *size-item*. Such a condition is reflected in the status code placed in *status-item*. Note that the test occurs before the time-out test in the case that *timeout* is zero or TEST ONLY is specified.
5. The WAIT statement places the thread ID of the sending or terminating thread in *thread-2*.
6. The size of the message sent is placed in *size-item*. The size is expressed in standard character positions (bytes).
7. The status of the WAIT statement is placed in *status-item*. The following values are possible (these approximate the standard file status codes):

   "00" Success - message received

   "10" Exception - thread does not exist or terminated

   "99" Exception - timed-out

8. If the WAIT statement is successful (as indicated in rule 7), it executes *statement-1*. If an exception condition exists, it executes *statement-2* instead.

WRITE Statement

The WRITE statement adds a record to a file.

General Format

Format 1

```
WRITE record-name [ FROM source ]

   [ {BEFORE} ADVANCING { number [LINE ] } ]
     {AFTER }           {        [LINES] }
                        {      PAGE      }
                        { mnemonic-name  }        (HP COBOL)
```

```
   [ AT {END-OF-PAGE} statement-1 ]
        {EOP        }

   [ NOT AT {END-OF-PAGE} statement-2 ]
            {EOP        }

   [ END-WRITE ]
```

Format 2

```
WRITE record-name [ FROM source ]

   [ INVALID KEY statement-1 ]

   [ NOT INVALID KEY statement-2 ]

   [ END-WRITE ]
```

Format 3

```
WRITE record-name [ FROM source ] WITH NO { CONTROL    }
                                          { CONVERSION }
```

Syntax Rules

1. *Record-name* is the name of a record associated with a file described in the File Section of the Data Division. The associated file may not be a sort file.
2. *Source* is a data item or literal. It may not share any storage area with *record-name*.
3. *Number* is an integer numeric literal or data item. It must be non-negative.
4. *Mnemonic-name* is a user-defined word that may be assigned to Special Names in the ADVANCING clause of the WRITE statement. (This is a feature of *HP COBOL*. For details, see section 4.3.2, “Special Names Paragraph,” in *Transitioning to ACUCOBOL-GT*.)
5. *Statement-1* and *statement-2* are imperative statements.
6. A Format 1 WRITE statement must be associated with a sequential file. A Format 2 WRITE statement must be associated with a relative or indexed file.

7. The words END-OF-PAGE and EOP are equivalent.
8. If the END-OF-PAGE phrase is used, the file description entry containing *record-name* must have a LINAGE clause.
9. The ADVANCING PAGE and END-OF-PAGE phrases cannot both be used in the same WRITE statement.
10. A Format 3 WRITE statement must be associated with a sequential file.

General Rules

1. The file associated with *record-name* must be open when the WRITE statement executes. For sequential access mode files, the file must be open in the OUTPUT or EXTEND modes. For random and dynamic access mode files, the file must be open in the OUTPUT, I-O, or EXTEND mode.
2. The WRITE statement adds the contents of *record-name* to the file according to the following rules:
    a. For sequential access mode files, the record is added to the end of the file. If the file is indexed, the record's primary key must contain a value that is larger than all of the primary keys currently in the file. If a relative file has a RELATIVE KEY data item specified for it, the record number of the added record is moved to this data item when the WRITE statement completes.
    b. For random and dynamic access mode files, the record is inserted into the file according to its key value. For relative files, the record is placed at the record number described by the file's RELATIVE KEY data item. For indexed files, the values of *record-name's* key items are used to insert the record in the file to maintain the correct key orderings.
3. If the FROM phrase is used, the *source* item is moved to *record-name* according to the rules of the MOVE statement before *record-name* is written to the file.
4. The FILE STATUS data item is updated by the WRITE statement.

5. Some sequential files are considered to be print files. A print file has page positioning information specified in it along with the record data. A file with the PRINT option of the ASSIGN clause specified for it is a print file. A file that is referenced by any WRITE statement that contains an ADVANCING phrase is also a print file.

6. If the ADVANCING phrase is specified, the file is treated as a print file and the following occurs:

    a. If *number* is positive, the representation of the printed page is advanced a number of lines equal to that value.

    b. If *number* is zero, no repositioning of the representation of the printed page is performed.

    c. If the PAGE phrase is used, the representation of the printed page is advanced to the next page boundary. If the associated file has a LINAGE clause specified for it, this is done by spacing the appropriate number of lines. Otherwise this is done by physically advancing the device to the top of the next physical page.

    d. If the BEFORE phrase is specified, the page advancement specified occurs after *record-name* is added to the file.

    e. If the AFTER phrase is used, the page advancement specified occurs before *record-name* is added to the file.

    f. If no ADVANCING phrase is specified, and the file is a print file, AFTER ADVANCING 1 LINE is implied.

7. The invalid key condition exists when any of the following occur:

    a. A relative file record is written in the random or dynamic access modes, and the record number indicated by the RELATIVE KEY data item is already used by another record.

    b. An indexed file record is written and the primary key value in *record-name* is used by another record already in the file.

    c. An alternate key value in *record-name* is already being used by a record in the file, and that alternate key does not allow for duplicates.

8. When the invalid-key condition exists, the WRITE statement is unsuccessful and the following occurs:

a. If the INVALID KEY phrase is specified, *statement-1* executes; otherwise

b. If an appropriate USE AFTER EXCEPTION procedure exists, that error procedure executes; otherwise

c. A message is printed and the program halts.

9. If the WRITE statement is successful and the NOT INVALID KEY phrase is specified, *statement-2* is executed.

10. The ordering of indexed file keys for alternate keys that allow duplicates is the order in which the records are written to the file for those duplicated values.

11. The current file position is not modified by the WRITE statement.

12. The logical size of the record written to the file is the size of *record-name*. The physical size may be different due to physical characteristics of the file.

13. If the file associated with *record-name* has a LINAGE clause, the following rules apply:

    a. An *automatic page overflow* condition occurs when the WRITE statement cannot be fully accommodated in the page body. This occurs when the WRITE statement would cause the LINAGE-COUNTER to exceed the number of lines in the page body specified by the LINAGE clause. When this happens, the line is presented before or after (depending on the phrase used) the device is positioned to the first line of the next logical page.

    b. An *end-of-page* condition occurs when the WRITE statement causes printing or spacing in the footing area of the page body. This occurs when the WRITE statement causes the LINAGE-COUNTER to equal or exceed the value of the FOOTING phrase of the associated LINAGE clause. If no FOOTING phrase is present, then the end-of-page condition cannot occur. Note that the end-of-page condition does not imply any automatic device positioning.

    c. If the END-OF-PAGE phrase is used, then *statement-1* executes if either an automatic page overflow or an end-of-page condition exists. Otherwise *statement-2* executes (if specified).

14. The WRITE statement removes trailing spaces from *record-name* if the file specifies trailing-space suppression. For related information, see:

    - General Rule number 7b (above)
    - **Section 4.3.1, "File-Control Paragraph"**
    - **Section 5.1.7, "File Types"** under "Text Records"
    - Appendix H, Book 4, under the subheading STRIP_TRAILING_SPACES

15. A Format 3 WRITE statement writes the data in record-name to its file without any additional carriage-control information. In addition, if the NO CONVERSION option is specified, no trailing spaces are removed from the record, even if they otherwise would be. Use this to send information to devices when carriage-control is inappropriate, for example, when sending a form to a laser printer.

16. Depending on the host environment, it is possible that records written with a Format 3 WRITE statement cannot later be retrieved with a READ statement.

For configuration variables related to the WRITE statement, see Appendix H, Book 4, *Appendices* under:

CARRIAGE_CONTROL_FILTER
COMPRESS_FACTOR
FLUSH_COUNT
FLUSH_ON_ACCEPT
MIN_REC_SIZE
PAGE_EJECT_ON_CLOSE
STRIP_TRAILING_SPACES
V_BUFFERS
V_BUFFER_DATA

XML GENERATE Statement

The XML GENERATE statement generates an XML document from existing, COBOL data (i.e., it translates COBOL data to XML format). It is an implementation of the IBM Enterprise COBOL verb of the same name and

is provided to simplify IBM migrations; however, any customer wishing to write XML data can use this verb. See *A Guide to Interoperating with ACUCOBOL-GT*, Chapter 11 for additional information on working with XML data.

General Format

```
XML GENERATE identifier-1 FROM identifier-2 [COUNT [IN] identifier-3]
  [[ON] EXCEPTION imperative-statement-1]
  [NOT [ON] EXCEPTION imperative-statement-2]
  [END-XML]
```

Syntax Rules

1. *identifier-1* is the receiving area for the XML document. It must be an alphanumeric data item, and it must not overlap *identifier-2* or *identifier-3*.

2. *identifier-1* must be large enough to contain the generated XML document. Typically, it should be from five to eight times the size of *identifier-2*, depending on the length of the data-name or data-names within *identifier-2*. If *identifier-1* is not large enough, an error condition exists at the end of the XML GENERATE statement.

3. *identifier-2* is the source of the data to be converted to an XML document. It must not overlap with *identifier-1* or *identifier-3*.

4. *identifier-3* is a numeric data item. It may not overlap with *identifier-1* or *identifier-2*.

General Rules

1. XML GENERATE ignores certain data items when they are specified by *identifier-2*. These include:

   - Unnamed elementary data items or elementary FILLER items.

   - Slack bytes inserted for SYNCHRONIZED items.

   - Data items subordinate to identifier-2 that are described by the REDEFINES clause, or that are subordinate to such a redefining item.

- Data items subordinate to identifier-2 that are described with the RENAMES clause.
- Group data items whose subordinate data items are all ignored.

2. All data items specified by *identifier-2* that are not ignored as defined above must satisfy the following conditions:
   - Each elementary data item must either have class alphabetic, alphanumeric, numeric, or be an index data item.
   - There must be at least one such elementary item.
   - Each non-FILLER data name must be unique within any immediately superordinate group data item.
3. The COUNT IN phrase indicates that the count of generated XML characters (in bytes) should be stored in *identifier-3*, the data count field. *identifier-3* must be an integer data item without the symbol "P" in its picture string. *identifier-3* must not overlap identifier-1 or identifier-2.
4. ON EXCEPTION phrase. When an error occurs during XML document generation, an exception condition exists. An example of this is when *identifier-1* is not large enough to contain the generated XML document. In this case, XML generation stops and the content of the receiver, *identifier-1*, is undefined. If the COUNT IN phrase was specified, *identifier-3* contains the number of character positions that were generated. This can range from zero to the length of *identifier-1*.

   If the ON EXCEPTION phrase is specified, control is transferred to *imperative-statement-1*. If it is not specified, NOT ON EXCEPTION phrases are ignored, and control is transferred to the end of the XML GENERATE statement.

At termination of an XML GENERATE statement, special register XML-CODE contains either “0”, indicating successful completion of XML generation, or a non-zero error code, indicating that an exception occurred during XML generation. Following are the possible exception codes that you may encounter:

Code	Description
400	The receiver was too small to contain the generated XML document. The COUNT IN data item, if specified, contains the count of character positions that were actually generated.
600 – 699	Internal error. Please report the error to your customer support analyst.

5. NOT ON EXCEPTION phrase. If no exception conditions arise during generation of the XML document, control is passed to *imperative-statement-2*, if specified, or to the end of the XML GENERATE statement. If an ON EXCEPTION phrase is specified, it is ignored. Special register XML-CODE contains a zero after the XML GENERATE statement has finished executing.

6. The END-XML phrase is an explicit scope terminator that delimits the scope of both XML GENERATE and XML PARSE statements. With END-XML, conditional XML GENERATE or XML PARSE statements can be nested in other conditional statements. Conditional XML GENERATE or XML PARSE statements specify the ON EXCEPTION or NOT ON EXCEPTION phrase.

   The scope of a conditional XML GENERATE or XML PARSE statement is terminated by:

   - An END-XML phrase at the same level of nesting
   - A separator period

Operation of XML GENERATE

Eligible elementary data items in *identifier-2* are converted to character format. (See "**Data conversion**" and "**Data trimming**" for details.) Only the first definition of each storage area is processed. Redefinitions are not included, nor are data items that are effectively defined by the RENAMES clause.

Once the data content is converted, it is inserted as element character content in XML markup. The XML element names are derived from the data-names in *identifier-2*. (See "**Element naming**" for more information.) The names of group items that contain the selected elementary items are retained as parent elements. No extra white space is inserted to make the generated XML more readable. An XML declaration is not generated.

If the receiving area specified by identifier-1 is not large enough to contain the resulting XML document, an error condition arises. See the **ON EXCEPTION phrase**, General Rule #4, for details.

Caution: If *identifier-1* is longer than the generated XML document, only the initial part of *identifier-1* changes. The rest of *identifier-1* contains the data that was present before this execution of the XML GENERATE statement. To avoid referring to that data, either initialize *identifier-1* to spaces before the XML GENERATE statement or specify the COUNT IN phrase.

Use the COUNT IN phrase to determine the total number of character positions, in bytes, that were generated. *identifier-3* will then contain this information after XML GENERATE executes. You can use *identifier-3* as a reference modification length field to refer to the part of *identifier-2* that contains the generated XML document.

After execution of the XML GENERATE statement, special register XML-CODE contains either zero, indicating successful completion, or a non-zero exception code.

Please note that the XML PARSE statement also uses special register XML-CODE. Therefore, if you code an XML GENERATE statement in the processing procedure of an XML PARSE statement, save the value of XML-CODE before that XML GENERATE statement executes and restore the saved value after the XML GENERATE statement terminates.

Data conversion

How elementary data items are converted to character format depends on the type of data item:

- Alphabetic, alphanumeric, alphanumeric-edited, external floating-point, and numeric-edited items are not converted.
- Fixed-point numeric data items are converted as if they were moved to a numeric-edited item that has:
  - An explicit decimal point, if the numeric item has at least one decimal position
  - The same number of decimal positions as the numeric item
  - A leading '-' picture symbol if the data item is signed and has an S in its PICTURE clause

  For COMPUTATIONAL-5 (COMP-5) binary data items, the number of integer positions depends on the number of '9' symbols in the picture character string. If the data item has one to four '9' picture symbols, the number of integer positions is five minus the number of decimal places. If the data item has five to nine '9' picture symbols, the number of integer positions is ten minus the number of decimal places. If the data item has 10 to 18 '9' picture symbols, the number of integer positions is 20 minus the number of decimal places.

  All other fixed-point numeric data items will have as many integer positions as the numeric item, but with at least one integer position.
- Internal floating-point data items are converted as if they were moved to a data item as follows:
  - For COMP-1: an external floating-point data item with PICTURE -9.9(8)E+99

- For COMP-2: an external floating-point data item with PICTURE -9.9(17)E+99 (illegal because of the number of digit positions)

- Index data items are converted as if they were declared USAGE COMP-5 PICTURE S9(9). After conversion, leading and trailing spaces and leading zeroes are removed, as described under "Data trimming."

After conversion, if a data item contains characters that are illegal in XML, the value in the data item before conversion or trimming is represented in hexadecimal, and an element tag name with the prefix "hex." is substituted for the regular tag name. For example, if data item Customer-Name is found at run time to contain LOW-VALUES, the XML element tag name 'hex.Customer-Name' is used instead of the normal 'Customer-Name', and the content is represented as a string of pairs of zero digits.

Any remaining instances of the five characters & (ampersand), ' (apostrophe), > (greater-than sign), < (less-than sign), and " (quotation mark) are converted into the equivalent XML references '&', ''', '>', '<', and '"', respectively.

Data trimming

Data values are trimmed after they are converted to character format. (Conversion is described under "Data conversion.") Values converted from signed numeric values have their leading space removed if the value is positive. Values converted from numeric items have leading zeroes eliminated (after any initial minus sign). This is up to but not including the digit immediately before the actual or implied decimal point. Trailing zeroes after a decimal point are retained. For example:

- -012.340 becomes -12.340.
- 0000.45 becomes 0.45.
- 0013 becomes 13.
- 0000 becomes 0.

Character values from alphabetic, alphanumeric data items have either trailing or leading spaces removed, depending on whether the corresponding data items have left or right justification, respectively--left being the default. Trailing spaces are removed from values whose corresponding data items do

not specify the JUSTIFIED clause. Leading spaces are removed from values whose data items do specify the JUSTIFIED clause. If a character value consists solely of spaces, all spaces are removed but one.

Element naming

The element tag names in the XML documents generated from *identifier-2* are derived from the name of the data item specified by *identifier-2* and from any eligible data-names that are subordinate to *identifier-2*. The following rules apply:

- The exact mixed-case spelling of data-names from the data description entry is retained. The spellings from any references to that data item (for example, in an OCCURS DEPENDING ON clause) are not used.
- Data-names beginning with a digit are prefixed by an underscore. For example, the data-name "4C" becomes XML tag name "_4C".
- Names of data items that contain characters that are illegal in XML version 1.0 are prefixed by "hex.", and the content itself is expressed in hexadecimal.

Nested XML GENERATE statements

When a given XML GENERATE statement appears as *imperative-statement-1* or *imperative-statement-2*, or as part of *imperative-statement-1* or *imperative-statement-2* of another XML GENERATE statement, that given XML GENERATE statement is a nested XML GENERATE statement.

Nested XML GENERATE statements are considered to be matched XML GENERATE and END-XML combinations proceeding from left to right. For this reason, when END-XML phrases are encountered, they are matched with the nearest preceding XML GENERATE statements that have not already been terminated.

XML PARSE Statement

The XML PARSE statement parses an XML document so that it can be processed by the COBOL program. It is an implementation of the IBM Enterprise COBOL verb of the same name and is provided to simplify IBM migrations; however, any customer wishing to read XML data can use this verb.

XML PARSE is similar to C$XML in that you parse (read) the XML data and move it into the appropriate working storage item. The difference is that with C$XML, if you know that the data lies in a certain element or attribute, you can retrieve that attribute directly. With XML PARSE, you set up a processing procedure so that when you encounter a new element or attribute, you can specify how and where you want to store that data.

See *A Guide to Interoperating with ACUCOBOL-GT*, Chapter 11 for additional information on working with XML data.

General Format

```
XML PARSE identifier-1 PROCESSING PROCEDURE [IS]
  procedure-name-1 [THROUGH procedure-name-2]
                   THRU
  [[ON] EXCEPTION imperative-statement-1]
  [NOT [ON] EXCEPTION imperative-statement-2]
  [END-XML]
```

Syntax Rules

1. *identifier-1* is an alphanumeric data item that contains the XML document character stream.

2. The PROCESSING PROCEDURE phrase specifies the name of a procedure to handle the various events that the XML parser generates.

3. *procedure-name-1, procedure-name-2* names a section or paragraph in the procedure division. Procedure-name-1 and procedure-name-2 must not name a procedure name in a declarative section.

4. *procedure-name-1* specifies the first (or only) section or paragraph in the processing procedure.

5. *procedure-name-2* specifies the last section or paragraph in the processing procedure.

General Rules

1. For each XML event, the parser transfers control to the first statement of the procedure named *procedure-name-1*. Control is always returned from the processing procedure to the XML parser. The point from which control is returned is determined as follows:

   - If *procedure-name-1* is a paragraph name and *procedure-name-2* is not specified, the return is made after the last statement of the *procedure-name-1* paragraph is executed.
   - If *procedure-name-1* is a section name and *procedure-name-2* is not specified, the return is made after the last statement of the last paragraph in the *procedure-name-1* section is executed.
   - If *procedure-name-2* is specified and it is a paragraph name, the return is made after the last statement of the *procedure-name-2* paragraph is executed.
   - If *procedure-name-2* is specified and it is a section name, the return is made after the last statement of the last paragraph in the *procedure-name-2* section is executed.

   This procedure is the same as if the COBOL program executed the PERFORM verb on the same paragraph(s).

2. *procedure-name-1* and *procedure-name-2* must define a consecutive sequence of operations to execute, beginning at the procedure named by *procedure-name-1* and ending with the execution of the procedure named by *procedure-name-2*.

   If there are two or more logical paths to the return point, then *procedure-name-2* can name a paragraph that consists of only an EXIT statement; all the paths to the return point must then lead to this paragraph.

3. The processing procedure consists of all the statements at which XML events are handled. The range of the processing procedure includes all statements executed by CALL, EXIT, GO TO, GOBACK, MERGE, PERFORM, and SORT statements that are in the range of the

processing procedure, as well as all statements in declarative procedures that are executed as a result of the execution of statements in the range of the processing procedure.

The range of the processing procedure must not cause any GOBACK or EXIT PROGRAM statement to be executed, except to return control from a program to which control was passed by a CALL statement that is executed in the range of the processing procedure.

The range of the processing procedure must not cause an XML PARSE statement to be executed, unless the XML PARSE statement is executed in an outermost program to which control was passed by a CALL statement that is executed in the range of the processing procedure.

A program executing on multiple threads can execute the same XML statement or different XML statements simultaneously. However, the compiler generates LOCK THREAD / UNLOCK THREAD statements immediately before or after the XML PARSE statement, so effectively only a single thread is executing during the entire execution of the XML PARSE.

The processing procedure can terminate the run unit with a STOP RUN statement.

For more details about the processing procedure, see "**Control Flow**."

4. ON EXCEPTION phrase. The ON EXCEPTION phrase specifies imperative statements to be executed when an exception condition is raised by XML PARSE.

   An exception condition exists when the XML parser detects an error while processing an XML document. The parser first signals the exception by passing control to the processing procedure with special register XML-EVENT containing the word, 'EXCEPTION'. The parser also provides a numeric error code in special register XML-CODE. Error codes are listed in the special register section.

   An exception condition also exists when the processing procedure sets XML-CODE to "-1" before returning to the parser for a normal XML event. This is done by the user to deliberately terminate parsing. In this case, the parser does not signal an XML exception event. If the ON EXCEPTION phrase is specified, control is transferred to

imperative-statement-1. If it is not specified, NOT ON EXCEPTION phrases are ignored, and control is transferred to the end of the XML PARSE statement. Special register XML-CODE contains the numeric error code for the XML exception or "-1" after execution of the XML PARSE statement. See "**Special Registers**" later in this section for details.

If the processing procedure handles the XML exception event and sets XML-CODE to zero before returning control to the parser, the exception condition no longer exists. If no other unhandled exceptions occur prior to the termination of the parser, control is transferred to *imperative-statement-2* of the NOT ON EXCEPTION phrase, if specified.

5. NOT ON EXCEPTION phrase. The NOT ON EXCEPTION phrase specifies imperative statements to be executed when no exception conditions exist at the conclusion of XML PARSE processing.

   When no exception conditions exist, control is transferred to *imperative-statement-2*, if specified, or to the end of the XML PARSE statement. If an ON EXCEPTION phrase is specified, it is ignored. Special register XML-CODE contains a zero after the XML PARSE statement has finished executing.

6. END-XML phrase. The END-XML phrase is an explicit scope terminator that delimits the scope of both XML GENERATE and XML PARSE statements. With END-XML, conditional XML GENERATE or XML PARSE statements can be nested in other conditional statements. Conditional XML GENERATE or XML PARSE statements specify the ON EXCEPTION or NOT ON EXCEPTION phrase.

   The scope of a conditional XML PARSE statement is terminated by:

   - An END-XML phrase at the same level of nesting
   - A separator period

Nested XML PARSE Statements

When a given XML PARSE statement appears as *imperative-statement-1* or *imperative-statement-2*, or as part of *imperative-statement-1* or *imperative-statement-2* of another XML PARSE statement, that given XML PARSE statement is a nested XML PARSE statement.

Nested XML PARSE statements are considered to be matched XML PARSE and END-XML combinations proceeding from left to right. For this reason, when END-XML phrases are encountered, they are matched with the nearest preceding XML PARSE statements that have not already been terminated.

Control Flow

When the XML parser receives control from an XML PARSE statement, it analyzes the XML document and transfers control to procedure-name-1 at the following points:

- At the start of the parsing process
- When a document fragment is found
- When the parser detects an error in parsing the XML document
- At the end of processing the XML document

Control returns to the XML parser when the end of the processing procedure is reached.

The exchange of control between the parser and the processing procedure continues until either:

- The entire XML document has been parsed, ending with the END-OF-DOCUMENT event.
- The parser detects an exception and the processing procedure does not reset special register XML-CODE to zero prior to returning to the parser.
- The processing procedure terminates parsing deliberately by setting XML-CODE to -1 prior to returning to the parser.

Then, the parser terminates and returns control to the XML PARSE statement with the XML-CODE special register containing the most recent value set by the parser or the processing procedure.

The XML-CODE, XML-EVENT, and XML-TEXT special registers contain information about each XML event passed to the processing procedure. The content of XML-CODE is defined during and after execution of an XML PARSE statement. The contents of all other XML special registers are undefined outside the range of the processing procedure.

For normal XML events, XML-CODE contains zero when the processing procedure receives control. For exception events, XML-CODE contains one of the exception codes specified later in this document. XML-EVENT is set to the event name, such as “START-OF-DOCUMENT”. XML-TEXT contains the piece of the document corresponding to the event, as described in XML-EVENT. For more information about the XML special registers, see “**Special Registers**” below.

For all kinds of XML events, if XML-CODE is not zero when the processing procedure returns control to the parser, the parser terminates without a further EXCEPTION event. Setting XML-CODE to “-1” before returning to the parser for an event other than EXCEPTION forces the parser to terminate with a user-initiated exception condition. For some EXCEPTION events, the processing procedure can handle the event, then set XML-CODE to zero to force the parser to continue, although subsequent results are unpredictable. When XML-CODE is zero, parsing continues until the entire XML document has been parsed or an exception condition occurs.

Special Registers

XML-CODE

When used in the XML PARSE statement, the XML-CODE special register is used to communicate status between the XML parser and the processing procedure.

For each event, the XML parser sets XML-CODE before transferring control to the processing procedure. It also does this at parser termination. You can reset XML-CODE before returning control to the parser.

The XML-CODE special register has the implicit definition:

```
01  XML-CODE PICTURE S9(9) USAGE BINARY VALUE 0.
```

When the XML parser encounters an XML event, it sets XML-CODE and then passes control to the processing procedure. For all events except EXCEPTION, XML-CODE contains zero when the processing procedure receives control.

For an EXCEPTION event, the parser sets XML-CODE to an exception code that indicates the nature of the exception. Exception codes are listed below. Note that these are different than IBM COBOL's exception codes.

XML PARSE Exception Code	**Description**
101	Out of memory
102	Syntax error in XML
103	No elements
104	Invalid token
105	Unclosed token
106	Partial character
107	Tag mismatch
108	Duplicate attribute
109	Junk after the doc element
110	Error in the parameter entity reference
111	Undefined entity
112	Recursive entity reference
113	Asynchronous entity
114	Bad character reference
115	Binary entity reference
116	Attribute external entity reference
117	Misplaced XML processing instructions
118	Unknown encoding
119	Incorrect encoding

XML PARSE Exception Code	Description
101	Out of memory
120	Unclosed cdata section
121	External entity handling required
122	Not standalone
123	unexpected error
124	entity declared in wrong place

If you want the parser to terminate after normal events without causing an EXCEPTION, set XML-CODE to "-1" before returning control to the parser. If you set XML-CODE to any other value, results are undefined. IBM customers should note that ACUCOBOL-GT ignores XML-CODEs of "0". This is because unlike the IBM COBOL parser, there are no exceptions that allow continuation of parsing in ACUCOBOL-GT. Our XML parser cannot continue once it has detected an error.

In ACUCOBOL-GT, no further events are returned from the parser. Control is passed to the statement that you specify in the ON EXCEPTION phrase, or to the end of the XML PARSE statement if you did not code an ON EXCEPTION phrase.

When the parser returns control to the XML PARSE statement, XML-CODE contains the most recent value set either by the parser or by the processing procedure.

XML-EVENT

The XML parser uses the XML-EVENT special register to communicate event information to the processing procedure. The information that is communicated is identified in the XML PARSE statement. Before passing control to the processing procedure, the XML parser sets XML-EVENT to the name of the XML event, as described in Table 1 at the end of this topic.

XML-EVENT has the implicit definition:

```
01  XML-EVENT USAGE DISPLAY PICTURE X(30) VALUE SPACE.
```

XML-EVENT cannot be used as a receiving data item.

XML-TEXT

The XML-TEXT special register is defined during XML parsing to contain document fragments that are of class alphanumeric. XML-TEXT is an elementary alphanumeric data item of the length of the contained XML document fragment. The length of XML-TEXT can vary from 0 through 16,777,215 bytes. There is no equivalent COBOL data description entry.

The parser sets XML-TEXT to the document fragment associated with an event before transferring control to the processing procedure when the operand of the XML PARSE statement is an alphanumeric data item.

Use the LENGTH function for XML-TEXT to determine the number of bytes that XML-TEXT contains.

XML-TEXT cannot be used as a receiving item.

Table 1. Contents of XML-EVENT and XML-TEXT Special Registers

XML event (content of XML-EVENT)	Content of XML-TEXT
ATTRIBUTE-CHARACTERS	The value within quotes or apostrophes. If the value includes an entity reference, this can be a substring of the attribute value.
ATTRIBUTE-NAME	The attribute name; the string to the left of "=".
COMMENT	The text of the comment between the opening character sequence "<!--" and the closing character sequence "-->".
CONTENT-CHARACTER	The single character corresponding with the predefined entity reference in the element content.
CONTENT-CHARACTERS	The element content between start and end tags. This can be a substring of the element content if the content contains an entity reference or another element.
DOCUMENT-TYPE-DECLARATION	The entire document type declaration including the opening and closing character sequences, "<!DOCTYPE" and ">".
ENCODING-DECLARATION	The value, between quotes or apostrophes, of the encoding declaration in the XML declaration.
END-OF-CDATA-SECTION	Always contains the string "]]>".

Table 1. Contents of XML-EVENT and XML-TEXT Special Registers

XML event (content of XML-EVENT)	Content of XML-TEXT
END-OF-DOCUMENT	Null, zero-length.
END-OF-ELEMENT	The name of the end element tag or empty element tag.
EXCEPTION	The part of the document successfully scanned, up to and including the point at which the exception was detected. Special register XML-CODE contains the unique error code identifying the exception.
PROCESSING-INSTRUCTION-DATA	The rest of the processing instruction, not including the closing sequence, “?>”, but including trailing, not leading, white space characters.
PROCESSING-INSTRUCTION-TARGET	The processing instruction target name that occurs immediately after the processing instruction opening sequence, “<?”.
STANDALONE-DECLARATION	The value between quotes or apostrophes of the stand-alone declaration in the XML declaration
START-OF-CDATA-SECTION	Always contains the string “<![CDATA[“.
START-OF-DOCUMENT	The entire document.
START-OF-ELEMENT	The name of the start element tag or empty element tag, also known as the element type.
VERSION-INFORMATION	The value between quotes or apostrophes of the version declaration in the XML declaration.

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z