# Contents

Notices ............................................................................................................. ix
Programming Interface Information ............................................................. ix
Trademarks and Service Marks ........................................................................ x

About This Manual .......................................................................................... xi
Who Should Use This Manual ........................................................................ xi
Industry Standards Used in Compiler Design .................................................. xiii

Chapter 1. An Introduction to the COBOL/400 Programming Language ........ 1
Extensions to the ANSI Standard ..................................................................... 1
Features of the COBOL/400 Compiler .............................................................. 2
Using COBOL/400 Syntax Notation ................................................................. 2
Reading the Syntax Diagrams ......................................................................... 3
Reading IBM Extensions ................................................................................. 5
CL Entry Codes ............................................................................................... 5
An Overview of COBOL/400 Programming .................................................... 6

Chapter 2. Entering Your Source Program on the AS/400 System ................. 9
Designing Your COBOL/400 Program ............................................................. 9
Entering Source Using SEU ............................................................................. 11

Chapter 3. Compiling a COBOL/400 Program ............................................... 15
Using the Create COBOL Program (CRTCBLPGM) Command ...................... 15
Using the CRTCBLPGM Prompt Displays ..................................................... 16
Entering CRTCBLPGM from the Command Line ......................................... 28
Entering CRTCBLPGM from a CL Program ................................................ 28
Syntax of the CRTCBLPGM Command ......................................................... 29
Compiling Your Source Program For the Previous Release .......................... 31
Using the PROCESS Statement to Specify Compiler Options ....................... 32
Understanding Compiler Output ...................................................................... 37
Specifying the Format of Your Listing ............................................................ 38
Browsing Your Compiler Listing Using SEU ................................................. 39
A Sample Program and Listing ..................................................................... 39

Chapter 4. Running Your COBOL Program .................................................. 51
Replying to Run-Time Inquiry Messages ...................................................... 52

Chapter 5. Debugging Your Program ............................................................. 55
Avoiding Common Coding Errors ................................................................ 56
Using Breakpoints ........................................................................................ 57
Example of a Program Using Breakpoints .................................................... 57
Changing Program Variables ....................................................................... 63
Considerations for Using Breakpoints .......................................................... 63
Using a Trace .................................................................................................. 64
Example of Using a Trace .............................................................................. 64
Considerations for Using a Trace .................................................................. 66
Using a Debug Run-Time Switch .................................................................. 67
Using a COBOL Formatted Dump ............................................................... 67

Chapter 6. COBOL/400 Exception and Error Handling ............................... 69

© Copyright IBM Corp. 1994
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Error Handling</td>
<td>69</td>
</tr>
<tr>
<td>Error Handling Overview</td>
<td>69</td>
</tr>
<tr>
<td>Using Error-Handling Application Programming Interfaces (APIs)</td>
<td>70</td>
</tr>
<tr>
<td>Internal and External File Status</td>
<td>70</td>
</tr>
<tr>
<td>General Error Detection</td>
<td>72</td>
</tr>
<tr>
<td>How File Status is Set</td>
<td>72</td>
</tr>
<tr>
<td>Message Monitor Generation</td>
<td>73</td>
</tr>
<tr>
<td>Ending of a COBOL Program</td>
<td>74</td>
</tr>
<tr>
<td>Return Codes</td>
<td>75</td>
</tr>
<tr>
<td>Standard and Nonstandard Error Handling Models</td>
<td>76</td>
</tr>
<tr>
<td>Effects of *STDERR and *NOSTDERR on File Status</td>
<td>79</td>
</tr>
<tr>
<td>Processing of I/O Verbs</td>
<td>80</td>
</tr>
<tr>
<td>Common Exceptions and Some of Their Causes</td>
<td>81</td>
</tr>
<tr>
<td>Recovery After a Failure</td>
<td>82</td>
</tr>
<tr>
<td><strong>Chapter 7. File and Data Management</strong></td>
<td>89</td>
</tr>
<tr>
<td>Device Independence and Device Dependence</td>
<td>89</td>
</tr>
<tr>
<td>Spooling</td>
<td>91</td>
</tr>
<tr>
<td>Output Spool</td>
<td>91</td>
</tr>
<tr>
<td>Input Spool</td>
<td>92</td>
</tr>
<tr>
<td>System Override Considerations</td>
<td>92</td>
</tr>
<tr>
<td>File and Record Locking by COBOL</td>
<td>93</td>
</tr>
<tr>
<td>Locking and Releasing Records</td>
<td>93</td>
</tr>
<tr>
<td>Sharing an Open Data Path</td>
<td>94</td>
</tr>
<tr>
<td>Commitment Control Considerations</td>
<td>94</td>
</tr>
<tr>
<td>Unblocking Input Records and Blocking Output Records</td>
<td>102</td>
</tr>
<tr>
<td>File Status and Feedback Areas</td>
<td>103</td>
</tr>
<tr>
<td>File Descriptions</td>
<td>104</td>
</tr>
<tr>
<td>Program-Described Files</td>
<td>104</td>
</tr>
<tr>
<td>Externally Described Files</td>
<td>105</td>
</tr>
<tr>
<td>Data Description Specifications (DDS)</td>
<td>106</td>
</tr>
<tr>
<td>Format 2 COPY Statement (DD, DDR, DDS, or DDDR Option)</td>
<td>112</td>
</tr>
<tr>
<td>Indicators</td>
<td>115</td>
</tr>
<tr>
<td>Data Structures Generated</td>
<td>115</td>
</tr>
<tr>
<td>Key Generation Examples</td>
<td>121</td>
</tr>
<tr>
<td>Declaring Data Items Using CVTOPT Data Types</td>
<td>130</td>
</tr>
<tr>
<td>DBCS-Graphic Fields</td>
<td>133</td>
</tr>
<tr>
<td>Variable-length DBCS-Graphic Fields</td>
<td>134</td>
</tr>
<tr>
<td>Examples</td>
<td>134</td>
</tr>
<tr>
<td>Cross-system Data Considerations</td>
<td>137</td>
</tr>
<tr>
<td><strong>Chapter 8. Transaction Files</strong></td>
<td>139</td>
</tr>
<tr>
<td>Program-Described Transaction Files</td>
<td>139</td>
</tr>
<tr>
<td>Externally Described Transaction Files</td>
<td>139</td>
</tr>
<tr>
<td>The Format 2 COPY Statement</td>
<td>139</td>
</tr>
<tr>
<td>Data Description Specifications</td>
<td>140</td>
</tr>
<tr>
<td>Processing an Externally Described Transaction File</td>
<td>142</td>
</tr>
<tr>
<td>Using Indicators with Transaction Files</td>
<td>142</td>
</tr>
<tr>
<td>Indicators in a Separate Indicator Area</td>
<td>143</td>
</tr>
<tr>
<td>Indicators in the Record Area</td>
<td>143</td>
</tr>
<tr>
<td>ASSIGN Clause and the Separate Indicator Area Attribute</td>
<td>143</td>
</tr>
<tr>
<td>Data Description Entry–Boolean Data</td>
<td>144</td>
</tr>
<tr>
<td>INDICATORS Phrase</td>
<td>145</td>
</tr>
<tr>
<td>Indicators in a Separate Indicator Area</td>
<td>145</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Handling Data Errors</td>
<td>267</td>
</tr>
<tr>
<td>Performance Considerations</td>
<td>268</td>
</tr>
<tr>
<td>PICTURE Clauses for Numeric Items</td>
<td>268</td>
</tr>
<tr>
<td>Eight-Byte Binary Items</td>
<td>268</td>
</tr>
<tr>
<td>Segmentation</td>
<td>268</td>
</tr>
<tr>
<td>Calling a COBOL Program from a Non-COBOL Program</td>
<td>268</td>
</tr>
<tr>
<td>Debugging</td>
<td>269</td>
</tr>
<tr>
<td>*NORANGE Option</td>
<td>269</td>
</tr>
<tr>
<td>*DUPKEYCHK Option</td>
<td>270</td>
</tr>
<tr>
<td>Relative Files</td>
<td>270</td>
</tr>
<tr>
<td>Indicators</td>
<td>270</td>
</tr>
<tr>
<td>Commitment Control</td>
<td>270</td>
</tr>
<tr>
<td>Reading without Record Locks</td>
<td>270</td>
</tr>
<tr>
<td>Initializing Variables</td>
<td>270</td>
</tr>
<tr>
<td>Blocking Records</td>
<td>270</td>
</tr>
<tr>
<td>Program Loops</td>
<td>271</td>
</tr>
<tr>
<td>Tracing a Loop in a Program</td>
<td>271</td>
</tr>
<tr>
<td>Errors That Can Cause a Loop</td>
<td>271</td>
</tr>
<tr>
<td>Chapter 12. Communicating Between Programs</td>
<td>273</td>
</tr>
<tr>
<td>Transferring Control to Another Program</td>
<td>273</td>
</tr>
<tr>
<td>Main Programs and Subprograms</td>
<td>273</td>
</tr>
<tr>
<td>Returning Control from a Called Program</td>
<td>274</td>
</tr>
<tr>
<td>Initialization of Storage</td>
<td>279</td>
</tr>
<tr>
<td>Calling Another Program</td>
<td>279</td>
</tr>
<tr>
<td>Passing Data Using BY REFERENCE or BY CONTENT</td>
<td>279</td>
</tr>
<tr>
<td>In the Linkage Section</td>
<td>281</td>
</tr>
<tr>
<td>Grouping Data to be Passed</td>
<td>281</td>
</tr>
<tr>
<td>Call by Identifier</td>
<td>282</td>
</tr>
<tr>
<td>Using Pointers in a COBOL/400 Program</td>
<td>282</td>
</tr>
<tr>
<td>Defining Pointers and Pointer Alignment</td>
<td>283</td>
</tr>
<tr>
<td>Pointers and the REDEFINES Clause</td>
<td>284</td>
</tr>
<tr>
<td>Reading and Writing Pointers</td>
<td>285</td>
</tr>
<tr>
<td>Initializing Pointers Using the NULL Figurative Constant</td>
<td>286</td>
</tr>
<tr>
<td>LENGTH OF Special Register</td>
<td>286</td>
</tr>
<tr>
<td>Setting the Address of Linkage Items</td>
<td>287</td>
</tr>
<tr>
<td>Using ADDRESS OF and the ADDRESS OF Special Register</td>
<td>287</td>
</tr>
<tr>
<td>Using Pointers in a MOVE Statement</td>
<td>287</td>
</tr>
<tr>
<td>Using Pointers in a CALL Statement</td>
<td>289</td>
</tr>
<tr>
<td>Using Pointers and APIs to Access User Spaces</td>
<td>291</td>
</tr>
<tr>
<td>Processing a Chained List</td>
<td>302</td>
</tr>
<tr>
<td>Data Areas</td>
<td>305</td>
</tr>
<tr>
<td>Local Data Area</td>
<td>305</td>
</tr>
<tr>
<td>Program Initialization Parameters (PIP) Data Area</td>
<td>306</td>
</tr>
<tr>
<td>File Considerations</td>
<td>306</td>
</tr>
<tr>
<td>Appendix A. Segmentation Feature</td>
<td>309</td>
</tr>
<tr>
<td>Segmentation Concepts</td>
<td>309</td>
</tr>
<tr>
<td>Program Segments</td>
<td>309</td>
</tr>
<tr>
<td>Segmentation Logic</td>
<td>310</td>
</tr>
<tr>
<td>Segmentation Control</td>
<td>310</td>
</tr>
<tr>
<td>COBOL Source Program Considerations</td>
<td>310</td>
</tr>
<tr>
<td>Segmentation–Special Considerations</td>
<td>312</td>
</tr>
</tbody>
</table>
Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM’s licensed program may be used. Any functionally equivalent product, program or service that does not infringe any of IBM’s intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user’s responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 208 Harbor Drive, Stamford, Connecticut, USA 06904-2501

Important changes or additions to the text are indicated by a vertical line (|) to the left of the change or addition.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Programming Interface Information

This publication is intended to help the customer write COBOL/400 programs.

This publication also documents General-Use Programming Interface and Associated Guidance Information.

General-Use programming interfaces allow the customer to write programs that obtain the services of the COBOL/400 compiler.

General-Use Programming Interface and Associated Guidance Information is identified where it occurs, either by an introductory statement to a chapter or section or by the following marking:

| General-Use Programming Interface |

General-Use Programming Interface and Associated Guidance Information...

| General-Use Programming Interface |

End of General-Use Programming Interface
**Trademarks and Service Marks**

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States and/or other countries:

<table>
<thead>
<tr>
<th>Term</th>
<th>IBM Corporation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application System/400</td>
<td>AS/400</td>
</tr>
<tr>
<td>COBOL/400</td>
<td>IBM</td>
</tr>
<tr>
<td>Operating System/2</td>
<td>Operating System/400</td>
</tr>
<tr>
<td>OS/400</td>
<td>RPG/400</td>
</tr>
<tr>
<td>SQL/400</td>
<td>System/370</td>
</tr>
<tr>
<td>400</td>
<td>Systems Application Architecture</td>
</tr>
</tbody>
</table>
About This Manual

This manual provides information an application programmer needs to write, compile, test, debug, and run COBOL/400 programs on the IBM Application System/400 (AS/400) system.

This manual refers to other IBM publications. These publications are listed in the “Bibliography” on page 383 with their full title and base order number. When they are referred to in text, a shortened version of the title is used.

Who Should Use This Manual

This manual is intended for programmers who have some experience with the COBOL programming language and for the operators who run the programs. It is a guide to programming in the COBOL/400 language for users of the AS/400 system. As a user, you should have a basic understanding of data processing concepts, the COBOL programming language, and the IBM Operating System/400 (OS/400) operating system.

Using this manual, you will be able to:

- Design COBOL/400 programs
- Code COBOL/400 programs
- Enter, compile, and run COBOL/400 programs
- Test and debug COBOL/400 programs
- Study coded COBOL/400 examples.

Note: You should be familiar with Chapters 1 through 4 of this manual before proceeding to the other chapters.

Use this manual with the COBOL/400 Reference, SC09-1813, which describes each component and feature of the COBOL/400 language. The COBOL/400 User’s Guide, SC09-1812 and the COBOL/400 Reference together describe the COBOL/400 compiler and language.

For information about the complete library of AS/400 documents, consult the Publications Guide, GC41-9678, which contains a brief description of the contents of each AS/400 manual.

Before you use this manual, you should be familiar with the following information:

- How to use the controls and indicators on your display and how to use the keys on your keyboard, such as:
  - Cursor movement keys
  - Function keys
  - Field exit keys
  - Insert and Delete keys
  - Error Reset key.

For information about your display station, refer to:

• How to operate your display station when it is linked to the IBM AS/400 system and running AS/400 software. This means knowing how to use the OS/400 operating system and its Control Language (CL) to do such things as:
  – Sign on and sign off the display station
  – Interact with displays
  – Use Help
  – Enter CL commands
  – Use Application Development Tools
  – Respond to messages
  – Perform file management.

• The Programming: Control Language Programmer’s Guide, SC41-8077 which contains the basic concepts of OS/400 CL functions.

To find out more about the operating system and its control language, refer to these IBM publications:
  – Advanced Backup and Recovery Guide, SC41-8079

• The Data Management Guide, SC41-9658 which provides information on using data management support to allow an application to work with files.

The manual includes information on:
  – Fundamental structure and concepts of data management support on the system
  – Data management support for display stations, printers, tapes, and diskettes, as well as spooling support
  – Overrides and file redirection (temporarily making changes to files when an application is run)
  – Copying files by using system commands to copy data from one place to another
  – Tailoring a system using double-byte data.

• How to use the following Application Development Tools:
  – The Screen Design Aid (SDA) is used to design and code displays. Information about this product is contained in Application Development Tools: Screen Design Aid User’s Guide and Reference, SC09-1340.
  – The Source Entry Utility (SEU) is a full-display editor you can use to enter and update your source members. Information about this product is contained in Application Development Tools: Source Entry Utility User’s Guide and Reference, SC09-1338.

• The Structured Query Language (SQL) allows you to insert SQL statements into COBOL/400 programs. Information about this product is contained in Systems Application Architecture* Structured Query Language/400 Reference, SC41-9608 and in Systems Application Architecture* Structured Query Language/400 Programmer’s Guide, SC41-9609

• The Customer Information Control System/400 (CICS/400*) licensed program allows you to enter transactions at remote work stations, and process them concurrently with user-written application programs. The licensed program
includes functions for building, using, and maintaining databases, and for communicating with CICS on other operating systems.

Information about using this product for application programming is contained in the CICS/400 Application Programming Guide. SC33-0822.

Industry Standards Used in Compiler Design

The COBOL/400 compiler is designed according to the following industry standards as understood and interpreted by IBM, as of September, 1987:

- The March 1986 Federal Information Processing Standards Publication (FIPS PUB 21-2) intermediate level. Additional support is provided for many high-level features.


The COBOL language is maintained by the Conference On DAta SYstems Languages (CODASYL).
Chapter 1. An Introduction to the COBOL/400 Programming Language

COmmon Business Oriented Language (COBOL) is a programming language that resembles English. As its name suggests, COBOL is especially efficient for processing business problems. It emphasizes describing and handling of data items and of input/output records; thus, it is well adapted for managing large files of data.

The COBOL/400 language delivers many elements of IBM Systems Application Architecture® (SAA®) Common Programming Interface (CPI) COBOL, and is the implementing product on the AS/400 system.

The COBOL/400 Compiler and Library is an IBM licensed program that accepts and runs COBOL programs that follow the ANSI X3.23-1985 (American National Standard Programming Language COBOL, ANSI X3.23-1985, ISO 1989-1985) standard. ANSI is an organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

Extensions to the ANSI Standard

To help you use COBOL on the AS/400 system, the COBOL/400 licensed program also includes a number of IBM extensions to the ANSI X3.23-1985 standard. Significant extensions include:

- TRANSACTION I/O: You can send or receive records from a work station.
- COPY: You can use externally described files.
- DATABASE I/O: You can use standard COBOL Environment and Data Division entries to specify file identification, field definitions, and data structures. Clauses have been added to the READ, WRITE, REWRITE, DELETE, and START verbs to support the AS/400 database.
- Extended data types: computational-3 (internal decimal or packed decimal), and computational-4 (binary) data types are supported.
- Boolean and pointer data types are supported.
- You have the option to use the apostrophe instead of a quotation mark.
- The compiler-directing statements SKIP1/2/3, EJECT, and TITLE are supported.
- Extended ACCEPT/DISPLAY: Provides support for field-level work station I/O.
- LIKE clause: You can define the characteristics of a data name by copying them from a previously-defined data name.
- Compiler listing suppression: You can selectively suppress portions of the compiler listing by using the *CBL or *CONTROL statement, or the SUPPRESS phrase of the COPY statement.
- Hexadecimal nonnumeric literals are supported.
Features of the COBOL/400 Compiler

The following language-independent features are available with the COBOL/400 compiler:

- **Syntax checking:**
  The Source Entry Utility (SEU) provides a COBOL syntax checker that checks for errors in lines of code as you enter or change them. Error messages are displayed, allowing you to correct errors before compilation time.

- **The cross-reference option:**
  - Provides a listing of each Data Division name and Procedure Division paragraph name
  - Indicates the statement numbers of each reference to the item.

- **Suppression of diagnostic messages below a user-specified level.**

- **The Federal Information Processing Standard (FIPS) flagger issues messages identifying obsolete or nonconforming language elements in the COBOL source program.** A **source program** is a set of instructions that is written in a programming language and must be translated to machine language before the program can be run.

- **SAA flagging to highlight the functions in your program that are not portable to other SAA COBOL environments.**

Using COBOL/400 Syntax Notation

In COBOL, basic formats are presented in a uniform system of syntax notation which is explained in the following paragraphs. This notation is designed to assist you in writing COBOL source statements.

- **COBOL keywords appear in uppercase letters; for example:**
  PARM1
  They must be spelled exactly as shown. If any required keyword is missing, the compiler considers it an error.

- **Variables representing user-supplied names or values appear in all lowercase letters; for example:**
  parmx

- **For easier text reference, some words are followed by a hyphen and a digit or a letter; for example:**
  identifier-1
  This suffix does not change the syntactical definition of the word.

- **Arithmetic and logical operators (+, -, *, /, **, >, <, =, >=, and <=) that appear in syntax formats are required.** These operators are special character reserved words. For a complete listing of reserved COBOL/400 words, see the “Reserved Words” section of the COBOL/400 Reference.

- **All punctuation and other special characters appearing in the diagram are required by the syntax of the format when they are shown; if you leave them out, an error occurs in the program.**
• You must write the required clauses and the optional clauses, (when used), in the order shown in the diagram unless the associated rules explicitly state otherwise.

Reading the Syntax Diagrams

Throughout this book, syntax is described using the structure defined below.

• Read the syntax diagrams from left to right, and from top to bottom, following the path of the line:
  ➤➤ Indicates the beginning of a statement. Diagrams of syntactical units other than statements, such as clauses, phrases and paragraphs, also start with this symbol.
  ➤➤ Indicates that the statement syntax is continued on the next line.
  ➤➤ Indicates that a statement is continued from the previous line.
  ➤➤ Indicates the end of a statement. Diagrams of syntactical units other than statements, such as clauses, phrases and paragraphs, also end with this symbol.

Note: Statements within a diagram of an entire paragraph do not start with ➤➤ and end with ➤➤ unless their beginning or ending coincides with that of the paragraph.

• Required items appear on the horizontal line (the main path). Optional items appear below the main path:

  ➤➤ STATEMENT  ──required item  ──optional item

• When you can choose from two or more items, they appear vertically, in a stack. If you must choose one of the items, one item of the stack appears on the main path. If choosing an item is optional, the entire stack appears below the main path:

  ➤➤ STATEMENT  ──required choice1  ──required choice2  ──optional choice1  ──optional choice2

• An arrow returning to the left above an item indicates that this item can be repeated:

  ➤➤ STATEMENT  ──repeatable item
A repeat arrow above a stack of required or optional choices indicates that you can make more than one choice from the stacked items, or repeat a single choice:

The following example shows how the syntax is used:

Format

where item-1 is:

1. The STATEMENT keyword must be specified and coded as shown.
2. This operand is required. Either identifier-1 or literal-1 must be coded.
3. The operand item-1 is optional. It can be coded or not, as required by the application. If coded, it can be repeated, with each entry separated by one or more blanks. Entry selections allowed for this operand are described at the bottom of the diagram.
4. The operand identifier-4 is optional. If specified it may be repeated with one or more blanks separating each entry. Each entry may be assigned the keyword ROUNDED.
5. In cases where multiple lines must be continued past the right margin, line order from top to bottom is preserved.
6. The ON keyword is optional to the keyword SIZE ERROR, which is optional itself. If SIZE ERROR is coded, then the operand imperative-statement is required.
7. The END-STATEMENT keyword can be coded to end the statement. It is not a required delimiter.


Reading IBM Extensions

An IBM extension generally adds to or contradicts a rule or restriction that immediately precedes it. The standard is presented first, because some programmers use the COBOL/400 language without IBM extensions. The extension is then presented for those who do use them.

IBM extensions within figures or tables are shown in boxes unless they are explicitly identified as extensions.

Clauses and statements illustrated within syntax diagrams that are COBOL/400 language extensions to ANSI X3.23-1985 COBOL are enclosed in double lines, as follows:

```
╔══════════════════════════╗
║ /SM590000────RECORD─┬─────┬──┬────┬──┬── ║
║ │EXTERNALLY-DESCRIBED-KEY─ ║
║ └─KEY─┘ └─IS─┘ │ ║
╚══════════════════════════╝
```

IBM Extension

COBOL/400 language extensions to ANSI X3.23-1985 COBOL that are part of the text description are enclosed in IBM Extension bars, like this paragraph.

```
COBOL/400 language extensions to ANSI X3.23-1985 COBOL that are part of the text description are enclosed in IBM Extension bars, like this paragraph.
```

End of IBM Extension

COBOL clauses and statements illustrated within syntax diagrams that are syntax checked, but are treated as documentation by the COBOL/400 compiler, are enclosed by asterisks, as follows:

```
******************************************************************************
* RESERVE integer *
* AREA *
* AREAS *
******************************************************************************
```

CL Entry Codes

The box that appears in the lower right corner of each CL syntax diagram contains the entry codes that specify the environment in which the command can be entered. The codes indicate whether or not the command can be:

- Used in a batch or interactive job (outside a compiled program; Job:B or I)
- Used in a batch or interactive compiled program (Pgm:B or I)
- Used in a batch or interactive REXX procedure (REXX:B or I)
- Used as a parameter for the CALL CL command, or passed as a character string to the system program QCMDEXC (Exec).
An Overview of COBOL/400 Programming

You follow four major steps or phases to build your COBOL/400 program:

- Entering your source program
- Compiling your source program
- Debugging your program
- Running your compiled program.

Entering Your COBOL Program

The Source Entry Utility (SEU) provides a special display that corresponds to the standard COBOL coding form to help you enter an accurate COBOL source program into the system. SEU also provides a COBOL syntax checker that checks each line for errors as you enter or change them. For information on entering your COBOL/400 source, refer to Chapter 2, “Entering Your Source Program on the AS/400 System.” For more information on using SEU, see the SEU User’s Guide and Reference.

Compiling Your COBOL Program

After you have entered the source program into the system, you need to compile the source program using the Create COBOL Program (CRTCBLPGM) command. The compiler is called to create a COBOL object program and a listing. An object program is a set of instructions in machine-usable form. The object program is produced by a compiler from a source program.

You can specify various compiler options by using the CRTCBLPGM command, or by using the PROCESS statement with the desired options. Any options specified in the PROCESS statement override the corresponding options on the CRT CBLPGM command. This process is explained in detail in Chapter 3, “Compiling a COBOL/400 Program.”

Debugging Your COBOL Program

The OS/400 operating system provides the following functions that you can use to test and debug your programs:

- Test library
- Breakpoints
- Traces.

The COBOL/400 compiler provides the following functions for program testing and debugging:

- Debugging features
- Formatted dump.

These features allow you to monitor specific program operations during run time. You must decide what to monitor and what information to retrieve for debugging purposes.

See Chapter 5, “Debugging Your Program” for more information on debugging features.
Running Your COBOL Program

You can run your COBOL program many ways, depending on how the program is written, and who is using it. You can run a COBOL program by calling it from a CL program, from an application program, from another high-level language program, or from a user-created command.

When your program has ended, the system returns control to whoever called the program.

For more information on running your program, see Chapter 4, “Running Your COBOL Program.”
Chapter 2. Entering Your Source Program on the AS/400 System

This chapter provides the information you need to enter your program. This chapter also briefly describes the tools and methodology necessary to complete this step.

There are two ways to enter a COBOL source program into the system:

- You can enter your source program using the Source Entry Utility (SEU). This is the method documented in this chapter.
- You can enter your source program from diskette or tape by using the OS/400 copy function.
  
  Refer to the CL Reference for more information on how to use the COPY function for entry of the source program from diskette or tape.

To enter your COBOL source program using SEU, enter the Start Source Entry Utility (STRSEU) command, and specify CBL for the TYPE parameter. Refer to the SEU User’s Guide and Reference for further information on the STRSEU command and using SEU.

Designing Your COBOL/400 Program

You can use the skeleton program, Figure 1 on page 10, as a model for developing readable and efficient COBOL programs. Note that not all the entries provided below are required; most are provided for informational purposes only.
The Identification Division 1 is the only division that must be included; all other divisions are optional.

The Environment Division 2 is made up of two sections: the Configuration Section 3, which describes the overall specifications of the source and object computers, and the Input-Output Section 4, which defines each file, and specifies information needed for transmission of data between an external medium and the COBOL program.

The Data Division 5 describes the files to be used in the program and the records contained within the files. It also describes any internal working-storage data items that are needed.

The Procedure Division 6 consists of optional declaratives, and procedures that contain sections and/or paragraphs, sentences, and statements.

Source File Format
The standard record length of your source files is 92 characters. These 92 characters are made up of a 6-character sequence number, a 6-character date-last-modified area, and an 80-character data field.

The COBOL/400 compiler supports an additional record length of 102; a field of 10 characters containing supplementary information is placed at the end of the record (positions 93-102). This information is not used by the COBOL compiler, but is placed on the extreme right of the compiler listing. You are responsible for placing information into this field. If you want to use this additional field, create a source file with a record length of 102.

IBM supplies a source file where you can store your source records if you do not want to create your own file. This file, named QLBSRC, is in library QGPL and has a record length of 92 characters.
Entering Source Using SEU

SEU provides special display formats for COBOL. They correspond to the COBOL Coding Form and are designed to help you enter your COBOL source programs. Figure 2 shows a display format, the relationship between the headings on the COBOL Coding Form, and the labels on the display; it also identifies where you enter the source code.

Figure 2. An SEU Display Format

For a complete description of how to enter a source program using SEU, refer to the SEU User’s Guide and Reference.

Using the COBOL Syntax Checker in SEU

To use the COBOL syntax checker in SEU, specify the TYPE(CBL) parameter of the STRSEU command. The COBOL syntax checker checks each line for errors as you enter new lines or change existing lines. Incorrect source statements are identified and error messages displayed, allowing you to correct the errors before compiling the program. Because the COBOL syntax checker checks only one statement at a time, independently of statements that precede or follow it, only syntax errors within the source data can be detected. No interrelational errors, such as undefined names and incorrect references to names, are detected. These errors are detected by the COBOL compiler when the program is compiled.

Any time a source line is entered or changed, up to 496 lines of source code can be syntax checked as one unit. The length of a single unit of syntax-checking is determined by extending from an entered or changed line as follows:

A unit of syntax-checking extends towards the beginning of the source member until the first source line, or a line that ends in a period is found.

A unit of syntax-checking extends towards the end of the source member until the last source line, or a line that ends in a period is found.

If this unit spans more than 496 source lines (not including comment lines), the system responds with an appropriate message.
If there is an error in a unit of syntax-checking, the entire unit is presented in reverse image. The message at the bottom of the display refers to the first error in the unit.

Syntax checking occurs line by line as you enter the source code. Error messages are generated by lines consisting of incomplete statements. These disappear when the statements are completed, as in the example:

```
ADD A
TO BCD.
```

An error message is generated after the first line is entered and disappears after the second line is entered, when the statement is completed. A COBOL sentence can span a maximum of 496 lines. Also, if a source line is entered or changed, up to 496 lines of source code can be syntax checked as one unit.

The following regulations apply to syntax checking for COBOL source functions:

- Source code on a line with an asterisk (*) or a slash (/) in column 7 is not syntax checked. An asterisk indicates a comment line; a slash indicates a comment line and page eject.
- No compiler options are honored during syntax checking.
  
  For example, the syntax checker accepts both quotation marks or apostrophes as nonnumeric delimiters provided they are not mixed within one unit of syntax checking. The syntax checker does not check if the delimiter is the one that will be specified in the CRTCBLCMP command for compiling COBOL source statements, or in the PROCESS statement.
- The first sentence following any of the paragraph headers listed below must begin on the same line as the paragraph header.
  
  ```
  PROGRAM-ID.
  AUTHOR.
  INSTALLATION.
  DATE-WRITTEN.
  DATE-COMPILED.
  SECURITY.
  SOURCE-COMPUTER.
  OBJECT-COMPUTER.
  SPECIAL-NAMES.
  ```
- Character replacement specified by the CURRENCY and DECIMAL-POINT clauses of the SPECIAL-NAMES paragraph is not honored during interactive syntax checking.
- When using the REPLACING Identifier-1 BY Identifier-2 clause of the COPY statement and when either identifier includes reference modification, SEU checks for matching parentheses only. For more information on reference modification, see Chapter 11, “COBOL/400 Programming Considerations.”

**Syntax for Structured Query Language (SQL) Statements**

The syntax for SQL statements embedded in a COBOL source program is:

```
EXEC SQL—sql-statement—END-EXEC.
```
If the member type for the source program is SQLCBL or CICSSQLCBL, when the COBOL syntax checker encounters an SQL statement, the statement is passed to the SQL syntax checker. If an error is detected, a message is returned.

If an SQL statement is encountered, and if the member type is not SQLCBL or CICSSQLCBL, a COBOL message is returned indicating that a COBOL statement is in error.

If there are errors in the embedded SQL statement as well as errors in the preceding COBOL statements, the SQL error message will only be displayed after the preceding COBOL errors are corrected.

For more information about SQL statements, refer to the SQL/400* Reference.

Syntax for Customer Information Control System (CICS) Statements
The syntax for CICS statements embedded in a COBOL source program is:

```
  ➔EXEC CICS—cics-statement—END-EXEC.➔
```

If the member type for the source program is CICSCBL or CICSSQLCBL, when the COBOL syntax checker encounters a CICS statement, the COBOL syntax checker checks for only basic syntax errors.

If a CICS statement is encountered, and if the member type is not CICSCBL or CICSSQLCBL, a COBOL message is returned indicating that a COBOL statement is in error.

For more information about CICS/400 statements, refer to the CICS/400 Application Programming Guide.
Chapter 3. Compiling a COBOL/400 Program

You need to compile the COBOL/400 source program to produce a usable object program. You do this using the Create COBOL Program (CRTCBLPGM) command. The result of the compilation is a COBOL object program and a listing.

You can specify various compiler options by using the CRTCBLPGM command, or from within the program using the PROCESS statement. Any options specified in the PROCESS statement override the corresponding options on the CRTCBLPGM command. The PROCESS statement is discussed later in “Using the PROCESS Statement to Specify Compiler Options” on page 32.

During compilation, the compiler checks the syntax of the COBOL source program line by line, and also checks the relationships between the lines.

Using the Create COBOL Program (CRTCBLPGM) Command

To compile a COBOL/400 source program into an object program, you must enter the CRTCBLPGM command. This calls the COBOL/400 compiler. You can use the CRTCBLPGM command interactively, or in batch jobs, or from other programs on the AS/400 system.

*Programming Note:* The number of entries in the Object Definition Table (ODT) and the amount of storage required by a COBOL program varies with the number and kinds of COBOL statements used in the program. A combination of these factors can cause the AS/400 internal size limits for the program to be exceeded. If this occurs, try using the *NOUNREF option of the GENOPT parameter. If the problem persists, the program must be rewritten.

When the *NOUNREF option is specified, only names that are referenced or are needed for data structuring are defined. This option is useful when the program contains many unreferenced identifiers.
If you do not specify CBL as the source member type, the compiler issues a warning.

If the Format 2 COPY statement is used in the program to access externally described files, the operating system provides information about the externally described files to the compiled program.

If the COBOL compiler stops, the message LBL9001 Compile failed. Program not created. is issued. You can use a control language program that can monitor for this exception by using the control language Monitor Message (MONMSG) command.

**Using the CRTCBLPGM Prompt Displays**

To enter the CRTCBLPGM command from the CRTCBLPGM prompt displays, type CRTCBLPGM and press F4 (Prompt) to show the first display. The parameters and options are described in the order they appear on these displays, on pages 18 through 27. The default values are explained first, and are underlined.

Each parameter on this display shows a default value. Move the cursor past the items where you want default values to apply. Type over any items to set different values or options. If you are unsure about the setting of a parameter value, type a question mark (?) in the first position of the field and press Enter, or F4 (Prompt), to receive more detailed information. The question mark must be followed by a blank.

Figure 3 shows the CRTCBLPGM prompt displays. When you see the first CRTCBLPGM prompt display, you see only the required parameters prompted. To see the additional parameters, press F10. You see the first display shown in Figure 3. To see the remainder of the parameters, as shown in the second and third displays in Figure 3, page forward.
Create COBOL Program (CRTCBLPGM)
Type choices, press Enter.

Program ................. +PGMID Name, +PGMID
Library ................. +CURLIB Name, +CURLIB
Source file ............. +QLBLSRC Name
Library ................. +LIBL Name, +LIBL, +CURLIB
Source member .......... +PGM Name, +PGM
Generation severity level . . . 29 0-29
Text 'description' ....... +SRCMBRTXT

Additional Parameters
Source listing options ...... +SOURCE, +NOSOURCE, +SRC...
Generation options ......... +NOLIST, +LIST, +NOXREF...
More...

F3=Exit  F4=Prompt  F5=Refresh  F12=Cancel  F13=How to use this display
F24=More keys

Create COBOL Program (CRTCBLPGM)
Type choices, press Enter.

Conversion options ........ +NOVARCHAR, +VARCHAR...
Message Limit:
   Number of messages ....... +NOMAX 1-9999, +NOMAX
   Message limit severity .... 29 0-29
Print file ................. +SYSPRNT Name
Library ................. +LIBL Name, +LIBL, +CURLIB
FIPS flagging ............. + for more values
   SAA flagging ........….. +NOFLAG +FLAG
Extended display options ... +DFRWRT, +NODFRWRT...
Flagging severity ......... 0 0-99
Replace program .......... +YES +NO, +YES
Target release ............ +CURRENT +PRV, V2R1M...
User profile .............. +USER +USER, +OWNER
More...

F3=Exit  F4=Prompt  F5=Refresh  F12=Cancel  F13=How to use this display
F24=More keys

Create COBOL Program (CRTCBLPGM)
Type choices, press Enter.

Authority ............... +LIBCRTAUT Name, +LIBCRTAUT, +ALL...
Compiler debugging dump: 1 1-65535, *
                        65535 1-65535
Intermediate text dump ... 0 0-31

F3=Exit  F4=Prompt  F5=Refresh  F12=Cancel  F13=How to use this display
F24=More keys

Figure 3. The CRTCBCLPGM Prompt Displays
Parameters of the CRTCLPGLM Command
A description of the parameters for the CRTCLPGLM command follows. The default values are explained first, and are underscored for identification. The parameters and options are described in the order they appear on the prompt displays.

All object names specified for the CRTCLPGLM command must follow AS/400 naming conventions: the names may be basic names, 10 characters in length, composed of alphanumeric characters, the first of which must be alphabetic; or the names may be quoted names, 8 characters in length, enclosed in double quotes.

If you want to relate these parameter descriptions to the CRTCLPGLM syntax diagram, refer to Figure 4 on page 29.

PGM Parameter:
Specifies the program name and library name for the COBOL program object you are creating. The possible values are:

*PGMID
The name for the program object is taken from the PROGRAM-ID paragraph in the COBOL source program.

program-name
Enter a name to identify the compiled COBOL program. If you specify a program name for this parameter, and run the compilation in batch mode, the first program in the batch job uses this name; any other programs use the name specified in the PROGRAM-ID paragraph in the source program.

The possible library values are:

*CURLIB
If you do not specify a library name, the system searches the library list to find the library where the source file is located.

library-name
Enter the name of the library where the source file is located.

source-file-name
Enter the name of the source file that contains the COBOL source to be compiled. This source file should have a record length of 92.

The possible library values are:

*LIBL
If you do not specify a library name, the current library is used. If you have not assigned a library as the current library, QGPL is used.

*CURLIB
The current library is used. If you have not assigned a library as the current library, QGPL is used.

library-name
Enter the name of the library where the source file is located.

SRCMBR Parameter:
Specifies the name of the member that contains the COBOL source to be compiled. You can specify this parameter only if the source file referenced in the SRCFILE parameter is a database file. The possible values are:

*PGM
If you specified a program name for the PGM parameter, the compiler looks for the source program in a member having the same name as the program, and creates an object program with the same name as the program and member.

If you did not specify a program name for the PGM parameter, the compiler looks for the program source in the first member of the database source file, and creates an object program using the name specified in the PROGRAM-ID paragraph.

source-file-name
Enter the name of the source file that contains the COBOL source to be compiled. This source file should have a record length of 92.

The possible library values are:

*LIBL
If you do not specify a library name, the current library is used. If you have not assigned a library as the current library, QGPL is used.

*CURLIB
The current library is used. If you have not assigned a library as the current library, QGPL is used.

library-name
Enter the name of the library where the source file is located.

SRCFILE Parameter:
Specifies the name of the source file that contains the COBOL source to be compiled. You can specify this parameter only if the source file referenced in the SRCFILE parameter is a database file. The possible values are:

*PGM
If you specified a program name for the PGM parameter, the compiler looks for the source program in a member having the same name as the program, and creates an object program with the same name as the program and member.

If you did not specify a program name for the PGM parameter, the compiler looks for the program source in the first member of the database source file, and creates an object program using the name specified in the PROGRAM-ID paragraph.

source-file-name
Enter the name of the source file that contains the COBOL source to be compiled. This source file should have a record length of 92.

The possible library values are:

*LIBL
If you do not specify a library name, the current library is used. If you have not assigned a library as the current library, QGPL is used.

*CURLIB
The current library is used. If you have not assigned a library as the current library, QGPL is used.

library-name
Enter the name of the library where the source file is located.
source-file-member-name
Enter the name of the member that contains the COBOL source.

GENLVL Parameter:
Specifies the severity level that determines if a program object is created. The severity level corresponds to the severity level of the messages produced during compilation of the program. If the severity level of error messages is greater than the value you specify, a program object is not created. For example, if you specify 19 for this parameter, a program object is not created if the severity level of any of the messages is 20 or greater.

The possible values are:

29 If errors occur with a severity level greater than 29, no program object is created.

severity-level
Specify a one or two-digit number, 0 through 29. If errors occur with a severity level greater than this level, no program object is created.

TEXT Parameter:
Allows you to enter text that briefly describes the program and its function.

*SRCMBRTXT
Use the same text for the program object as that which describes the database file member containing the COBOL source. If the source comes from a device or in-line file, specifying *SRCMBRTXT has the same effect as specifying *BLANK.

*BLANK
No text is specified.

text-description
Enter the text that briefly describes the program and its function. The text can be a maximum of 50 characters in length and must be enclosed in apostrophes. The apostrophes are not part of the 50-character string.

OPTION Parameter:
Specifies the options to use when the COBOL source is compiled. The possible values are:

*SOURCE or *SRC
The compiler produces a source listing, consisting of the COBOL source input and all compilation-time error messages.

*NOSOURCE or *NOSRC
The compiler does not produce the source part of the listing. If you do not require a source listing, you should use this option because compilation may take less time.

*NOXREF
The compiler does not produce a cross-reference listing for the source program.

*XREF
The compiler produces a cross-reference listing for the source program.

*GEN
The compiler creates a program object after the program is compiled.

*NOGEN
The compiler does not create a program object after the source program is compiled. You might specify this option if you want only error listings at this time.

*NOSEQUENCE
The reference numbers are not checked for sequence errors.

*SEQUENCE
The reference numbers are checked for sequence errors. Sequence errors do not occur if the *LINENUMBER option is specified.

*NOVBSUM
Verb usage counts are not printed.

*VBSUM
Verb usage counts are printed.

*NONUMBER
The source-file sequence numbers are used for reference numbers.

*NUMBER
The user-supplied sequence numbers (columns 1 through 6) are used for reference numbers.
**LINENUMBER**

The sequence numbers created by the compiler are used for reference numbers. This option combines program source code and source code introduced by COPY statements into one consecutively numbered sequence. Use this option if you specify FIPS (Federal Information Processing Standards) flagging or SAA flagging.

**NOSECLVL**

Second level message text is not listed for this compilation.

**SECLVL**

Second level message text is listed for this compilation, along with the first-level error text.

**Note:** The first-level error text is printed each time the error occurs.

**NOMAP**

The compiler does not list the Data Division map.

**MAP**

The compiler lists the Data Division map.

**NOOPTIONS**

Options in effect are not listed for this compilation.

**OPTIONS**

Options in effect are listed for this compilation.

**QUOTE**

Specifies that the delimiter quotation mark (") is used for nonnumeric literals and Boolean literals. This also specifies that the value of the figurative constant QUOTE has the EBCDIC value of a quotation mark.

**Note:** Boolean data is a category of data items that are limited to a value of 1 or 0. A Boolean literal is a literal composed of a Boolean character enclosed in quotation marks and preceded by a B; for example: B"1".

**APOST**

Specifies that the delimiter apostrophe (') is used for nonnumeric literals and Boolean literals. This also specifies that the value of the figurative constant QUOTE has the EBCDIC value of an apostrophe.

**PRTCORR**

The compiler inserts comment lines in the compiler listing indicating which elementary items were included as a result of the use of the CORRESPONDING phrase.

**NOPRTCORR**

The compiler does not insert comment lines in the compiler listing when the CORRESPONDING phrase is used.

**NOSRCDBG**

This option determines the kind of information you see on your programmable workstation when using the CoOperative Development Environment/400 product to compile your COBOL programs. See the note on page 21 for further information.

The compiler does not produce source-level debugging information. If *NOLSTDBG is also in effect, the compiler does not produce source-level error information either.

**SRCDBG**

This option determines the kind of information you see on your programmable workstation when using the CoOperative Development Environment/400 product to compile your COBOL programs. See the note on page 21 for further information.

The compiler produces source-level error information and source-level debugging information.

You cannot specify *SRCDBG and *LSTDBG together. Specify one or the other.
**NOLSTDBG**
This option determines the kind of information you see on your programmable work station when using the CoOperative Development Environment/400 product to compile your COBOL programs. See the note on page 21 for further information.

The compiler does not produce a listing view, source-level error information, or listing-level debugging information.

**LSTDBG**
This option determines the kind of information you see on your programmable work station when using the CoOperative Development Environment/400 product to compile your COBOL programs. See the note on page 21 for further information.

The compiler produces a listing view, and listing-level debugging information. If *NOSRCDBG is also in effect, the compiler does not produce source-level error information either.

You cannot specify *SRCDBG and *LSTDBG together. Specify one or the other.

**Note:** You can only use the *NOSRCDBG, *SRCDBG, *NOLSTDBG and *LSTDBG options if you are using the AD/Cycle CoOperative Development Environment/400 product to compile your program. If you specify one or more of these options but do not have the CODE/400 product installed, the COBOL/400 compiler will not continue processing and an error message is issued. For more information on these options, see the CODE Debug Tool User's Guide and Reference, SC09-1622.

**PRINT**
The compiler produces a spool listing.

**NOPRINT**
The compiler does not produce a spool listing.

**NOATR**
Does not list the attributes for the IRP source.

**ATR**
Lists the attributes for the IRP source.

**RANGE**
At runtime, the system verifies that subscripts are within the correct ranges, but does not verify index ranges. It also
checks for reference modification and compiler-generated substring operations.

*RORANGE
Does not verify ranges at run-time.

Note: The *RANGE option generates code for checking subscript ranges. For example, it ensures that you are not attempting to access the 21st element of a 20-element array.

The *NORANGE option does not generate code to check subscript ranges.

These options do not eliminate the zero subscript checking performed by the operating system. If zero subscripts occur, the operating system will not permit their use and issues message MCH0603.

*UNREF
Includes unreferenced data items in the compiled program.

*NOUNREF
Does not include unreferenced data items in the compiled program. This reduces the number of ODT (object definition table) entries used, allowing a larger program to be compiled. The unreferenced data items still appear in the cross-reference listings produced by specifying OPTION (*XREF).

*NOOPTIMIZE
The compiler performs only standard optimizations for the program.

*OPTIMIZE
The compiler attempts to create a program that operates more efficiently and uses less storage. However, specifying *OPTIMIZE can substantially increase the time required to compile a program.

*NODDSFILLER
If no matching fields are found by a COPY DDS statement, no field descriptions are generated.

*DDSFILLER
When no matching fields are found by a COPY DDS statement, a single character FILLER field description, "07 FILLER PIC X", is always created.

*NOSYNC
The SYNCHRONIZED clause is syntax checked.

*SYNC
The SYNCHRONIZED clause causes the alignment of an elementary item on a natural boundary in storage.

*NOCRTF
Files that are unavailable at the time of an OPEN operation are not created dynamically.

*CRTF
Files that are unavailable at the time of an OPEN operation are created dynamically. When created, the file will inherit authority from the job profile. (See the discussion of the OPEN statement in the COBOL/400 Reference manual for the conditions under which dynamic file creation can occur.)

Note: The maximum record length for a file that will be created dynamically is 32,766.

*NODUPKEYCHK
Does not check for duplicate keys for INDEXED files.

*DUPKEYCHK
Checks for duplicate keys for INDEXED files. (See the discussion of the READ statement in the COBOL/400 Reference manual for the conditions under which the existence of records with duplicate keys will be signalled to a program.

Warning: Specifying this option can result in a loss in compiler performance.

*STDERR
Standard error handling is used. See Chapter 6, "COBOL/400 Exception and
The BLOCK CONTAINS clause controls the number of records to be blocked.

When no BLOCK CONTAINS clause is specified, the compiler allows blocking only of SEQUENTIAL access files with no START statement. The operating system determines the number of records to be blocked.

*STDINZ
The compiler initializes user data items to system defaults, provided that the items are not subject to a VALUE clause.

*NOSTDINZ
For those user items with no VALUE clause, the compiler does not initialize data items to system defaults.

*FS21DUPKY
The compiler reports a file status of 21 when processing an indexed file with duplicate keys in random or dynamic access mode, if the value of the key is changed between the mandatory READ statement and a following REWRITE or DELETE statement.

*NOFS21DUPKY
The compiler does not report a file status of 21 when processing an indexed file with duplicate keys in random or dynamic access mode. A REWRITE statement can change the key of a record.

CVTOPT Parameter:
Specifies how the compiler handles SAA date, time, and timestamp data types, DBCS-graphic data types, and variable-length character fields passed from externally-described files to your program through COPY DDS. The possible values are:

*NOVARCHAR
Variable-length fields are ignored, and are declared as FILLER fields.

*VARCHAR
Variable-length fields are declared as fixed-length group items, and are accessible to the program. For more information on variable-length fields, refer to
Declaring Data Items Using CVTOPT

Data Types

*NODATETIME
Date, time, and timestamp data types are ignored, and are declared as FILLER fields.

*DATETIME
Date, time, and timestamp data types are declared as fixed-length character fields, and are accessible to the program.

*NOGRAPHIC
DBCS-graphic data types are ignored, and are declared as FILLER fields.

*GRAPHIC
Fixed-length DBCS-graphic data types are declared as fixed-length alphanumeric fields, and are accessible to the program.

When the *VARCHAR option is also in use, variable-length DBCS-graphic data types are declared as fixed-length group items, and are accessible to the program. For more information on DBCS-graphic data types, refer to “DBCS-Graphic Fields” on page 133.

MSGLMT Parameter:
Controls compilation by indicating the maximum number of error messages of a given error severity level that can occur before compilation stops.

For example, you can stop compilation if more than three errors with a severity level of 20 or higher occur. In this example, you would specify 3 for the maximum number of error messages, and 20 for the maximum error severity level. If three errors of severity level 20 or higher occur, compilation continues, but when a fourth is encountered, compilation stops. If no messages equal or exceed the maximum severity level, compilation continues regardless of the number of errors encountered.

message-limit
The possible values for the maximum number of error messages are:

*NOMAX
Compilation continues until normal completion regardless of the number of errors encountered.

1-9999
Compilation stops if the number of errors of the specified severity level or higher exceeds the number you specify. If no messages equal or exceed the maximum severity level, compilation continues regardless of the number of errors encountered.

message-severity
The possible values for the maximum error severity level are:

29
Compilation stops if the number of errors with severity level 29 or higher exceeds the maximum number of error messages specified.

maximum-severity-level
Specify a one or two-digit number, 0 through 29. Compilation stops if the number of errors with the specified severity level or higher exceeds the maximum number of error messages you specify.

PRTFILE Parameter:
Specifies the name of the file to which the compiler listing is directed and the library where the file is located. The file should have a minimum record length of 132. If a file with a record length less than 132 is specified, information is lost.

The possible values are:

QSYSPR
If you do not specify a file name, the compiler listing is directed to QSYSPR, an IBM-supplied file.

file-name
Enter the name of the file to which the compiler listing is directed.

The possible library values are:

*LIBL
If a library-name is not specified, the system searches the library list, *LIBL, to find the library where the file is located.
The current library is used. If you have not assigned a library as the current library, QGPL is used.

library-name

Enter the name of the library where the file is located.

FLAGSTD Parameter:
Specifies the options for FIPS flagging.
(Select the *LINENUMBER option to ensure that the reference numbers used in the FIPS messages are unique.) The possible values are:

*NOFIPS
The source program is not FIPS flagged.

*MINIMUM
FIPS flag for minimum subset and higher.

*INTERMEDIATE
FIPS flag for intermediate subset and higher.

*HIGH
FIPS flag for high subset.

*NOSEG
The optional module SEGMENTATION is not FIPS flagged.

*SEG1
FIPS flag for optional module SEGMENTATION level 1 and higher.

*SEG2
FIPS flag for optional module SEGMENTATION level 2.

*SAAFLAG Parameter:
Specifies if you want flagging of COBOL/400’ functions that are not supported by SAA COBOL. (Select the *LINENUMBER option to ensure that the reference numbers used in the SAA COBOL messages are unique.) The possible values are:

*NOFLAG
SAA COBOL flagging is not performed.

*FLAG
SAA COBOL flagging is performed.

EXTDSPOPT Parameter:
Specifies the options to use for extended ACCEPT and extended DISPLAY statements for work station I/O. The possible values are:

*DFRWRT
Extended DISPLAY statements are held in a buffer until an extended ACCEPT statement is encountered, or until the buffer is filled.

If an extended ACCEPT statement is not encountered before the buffer is filled, the contents of the buffer are written to the display. When an extended ACCEPT statement is encountered, the current contents of the buffer are written to the display.

*NODFRWRT
Each extended DISPLAY statement is performed as it is encountered.

*UNDSPCHR
Displayable and undisplayable characters are handled by extended ACCEPT and extended DISPLAY statements.

*NOUNDSPCHR
Use this option when the data to be displayed contains extended DBCS characters. Only displayable characters are handled by extended ACCEPT and extended DISPLAY statements.
Although you must use this option for display stations attached to remote 3174 and 3274 controllers, you can also use it for local work stations. If you do use this option, your data must contain displayable characters. If the data contains values less than hexadecimal 20, the results are not predictable, ranging from unexpected display formats to severe errors.

*ACCUPDALL
All types of data are predisplayed in the extended ACCEPT statements regardless of the existence of the UPDATE phrase.

*ACCUPDNE
Only numeric edited data are predisplayed in the extended ACCEPT statements that do not contain the UPDATE phrase.

FLAG Parameter:
Specifies the minimum severity level of messages to be printed. The possible values are:

0 All messages are printed.

severity-level
Enter a one or two-digit number that specifies the minimum severity level of messages to be printed. Messages that have severity levels of the specified value or higher are listed.

REPLACE Parameter:
Specifies if a new program object is created when a program object of the same name in the same library already exists. The possible values are:

*YES
A new program object is created and any existing program object of the same name in the specified library is moved to library QRPLOBJ.

*NO
A new program object is not created if a program object of the same name already exists in the specified library.

TGTRLS Parameter:
Specifies the release of the operating system on which you intend to use the object being created. You can specify an exact release level in the format VxRxMx, where Vx is the version, Rx is the release, and Mx is the modification level. For example, V2R2M0 is version 2, release 2, modification level 0.

Note: To use the object on the target system, you must save the object to the target release level specified on the create command and then restore it on the target system.

The possible values are:

*CURRENT
The object is to be used on the release of the operating system currently running on your system. You can also use the object on a system with any subsequent release of the operating system installed.

*PRV
The object is to be used on the previous release with modification level 0 of the operating system. You can also use the object on a system with any subsequent release of the operating system installed.

release-level
Specify the release in the format VxRxMx. The object can be used on a system with the specified release or with any subsequent release of the operating system installed.

Valid values depend on the current version, release, and modification level, and they change with each new release.

USRPRF Parameter:
Specifies the user profile that will run the compiled COBOL program. The profile of the program owner or the program user is used to run the program and control which objects can be used by the program (including the authority the program has for each object). This parameter is not updated if the program already exists. To change the value of USRPRF, delete the program and recompile using the correct value.

The possible values are:

*USER
The user profile of the program user is to be used when the program is run.

*OWNER
The user profiles of both the program’s owner and user are to be used when the program is run. The collective sets of
object authority in both user profiles are to be used to find and access objects during the running of the program. Any objects that are created during the program are owned by the program's user.

**Note:** Specify the USRPRF parameter to reflect the security requirements of your installation. The security facilities available on the AS/400 system are described in detail in the *Security Reference*.

**AUT Parameter:**
Specifies the authority given to users who do not have specific authority to the program object, who are not on the authorization list, or whose group has no specific authority to the program object. You can alter the authority for all users, or for specific users after the program object is created by using the GRTOBJAUT (Grant Object Authority) or RVKOBJAUT (Revoke Object Authority) commands.

The possible values are:

* **LIBCRTAUT**
  The public authority for the object is taken from the CRTAUT keyword of the target library (the library that is to contain the created program object). This value is determined when the program object is created. If the CRTAUT value for the library changes after the program object is created, the new value does NOT affect any existing objects.

* **ALL**
  Provides authority for all operations on the program object except those limited to the owner or controlled by object authority and object management authority. The user can change the object and perform basic functions on it, such as running and debugging the program object.

* **USE**
  Provides object operational authority and read authority; authority for basic operations on the program object such as running the program. The user is prevented from changing the object.

* **EXCLUDE**
  The user cannot access the program object.

**authorization-list-name**
Enter the name of an authorization list of users and authorities to which the program is added. The program object is secured by this authorization list, and the public authority for the program object is set to *AUTL*. The authorization list must exist on the system when the CRTCBLPGM command is issued. Use the Create Authorization List (CRTAUTL) command to create your own authorization list.

**Note:** Specify the AUT parameter to reflect the security requirements of your installation. The security facilities available on the AS/400 system are described in detail in the *Security Reference*.

**DUMP Parameter:**
An IBM COBOL/400 debugging aid for IBM service personnel.

**ITDUMP (n) Parameter:**
An IBM debugging aid provided for IBM service personnel. This parameter makes the compiler dump the internal text at certain times during the compilation of the source program.
Entering CRTCBLPGM from the Command Line

You can enter the CRTCBLPGM command from the command line. Type CRTCBLPGM followed by the appropriate parameters to compile your program. Refer to the Figure 4 on page 29 for the correct syntax. If you are unsure about the parameters and their meanings, refer to the parameter and option descriptions on pages 18 through 27. Refer to the following examples of the syntax you would use to enter the CRTCBLPGM command from the command line.

Example 1
CRTCBLPGM SRCFILE(QGPL/QLBLSRC) SRCMBR(SAMPLE) SAAFLAG(+FLAG)

Partial Source for Member SAMPLE

ID DIVISION.
PROGRAM-ID. EXAMPLE.

The preceding example creates a COBOL/400 program from the source member SAMPLE in file QLBLSRC and library QGPL. The resulting program is called EXAMPLE. Specifying *FLAG for the SAAFLAG parameter tells the compiler to identify any functions that are not supported by SAA COBOL. In this example, all parameter defaults were used with the exception of the SRCFILE, SRCMBR, and SAAFLAG parameters.

Example 2
CRTCBLPGM PGM(TEST) SRCFILE(SOURCE1) CVTOPT(+GRAPHIC)

In the preceding example, the compiler looks for the file SOURCE1 in the library list, and looks for the member called TEST within that file. (The value for the SRCMBR parameter defaulted to *PGM, specifying to look for a member with the same name as the program to be created.) The compiler creates a COBOL/400 program called TEST from the source program found in the member TEST in the file SOURCE1. Specifying *GRAPHIC for the CVTOPT parameter indicates that if the DDS contains DBCS-graphic data types, you want the COBOL program to be able to reference them as alphanumeric fields.

Entering CRTCBLPGM from a CL Program

When you issue the CRTCBLPGM command from a CL program, you can use concatenation expressions for all parameter values. See the CL Reference for more information about concatenation expressions. Also, see the CL Reference for a detailed description of OS/400 object naming rules and for a complete description of OS/400 command syntax.

General-Use Programming Interface

You can use this command in QCMDEXC.

End of General-Use Programming Interface
Syntax of the CRTCLPGM Command

Figure 4 shows the syntax of the CRTCLPGM command.

```
CRTCBLPGM  PGM          (PGMID)
       └─SM590000─CRTCBLPGM─PGM─(─┬─/c5197CURLIB
                                    └─PGMID
                                    └─/c5197LIBL
                                ─┬─/c5197PGMID
                                └─/c5197LIBL
                               ──┐ └─/c5197PGMID
                              └─/c5197LIBL
                             ──┐ └─/c5197PGMID
                            └─/c5197LIBL
                           ──┐ └─/c5197PGMID
                          └─/c5197LIBL
                         ──┐ └─/c5197PGMID
                        └─/c5197LIBL
                       ──┐ └─/c5197PGMID
                      └─/c5197LIBL
                     ──┐ └─/c5197PGMID
                    └─/c5197LIBL
                   ──┐ └─/c5197PGMID
                  └─/c5197LIBL
                 ──┐ └─/c5197PGMID
                └─/c5197LIBL
               ──┐ └─/c5197PGMID
              └─/c5197LIBL
             ──┐ └─/c5197PGMID
            └─/c5197LIBL
           ──┐ └─/c5197PGMID
          └─/c5197LIBL
         ──┐ └─/c5197PGMID
        └─/c5197LIBL
       ──┐ └─/c5197PGMID
      └─/c5197LIBL
     ──┐ └─/c5197PGMID
    └─/c5197LIBL
   ──┐ └─/c5197PGMID
  └─/c5197LIBL
   └─/c5197PGMID
```

Figure 4 (Part 1 of 5). Syntax of the CRTCLPGM Command
Figure 4 (Part 2 of 5). Syntax of the CRTCLPBLGCM Command

Figure 4 (Part 3 of 5). Syntax of the CRTCLPBLGCM Command

Figure 4 (Part 4 of 5). Syntax of the CRTCLPBLGCM Command

Figure 4 (Part 5 of 5). Syntax of the CRTCLPBLGCM Command
You can compile a COBOL/400 program on an AS/400 system using the current release of the OS/400 operating system and restore it on an AS/400 system that uses a previous release of the operating system.

The Target Release (TGTRLS) parameter of the CRTCBLPGM command allows you to specify the release level on which you intend to use the object program. The TGTRLS parameter has three possible values: *CURRENT, *PRV and release-level:

- Specify *CURRENT if the object program is to be used on the release of the operating system currently running on your system. For example, if V2R2M0 is running on the system, *CURRENT means you intend to use the program on a system with V2R2M0 installed. This value is the default.
- Specify *PRV if the object program is to be used on the previous release with modification level 0 of the operating system. For example, if V2R2M0 is running on your system, *PRV means you intend to use the program on a system with V2R1M0 installed.
- release-level allows you to specify the release level on which you intend to use the object program. The values you can enter for this parameter depend on the current version, release, and modification level, and they change with each new release.

For more information about the TGTRLS parameter, see page 26.

You should be aware of the following limitations:

- Support to compile for use with the previous release is only available when you use the TGTRLS parameter of the CRTCBLPGM command. You must specify *PRV or the release level when you compile the program; you must then save the program, using the Save Object (SAVOBJ) or the Save Library (SAVLIB) CL command, in order to successfully restore it to the previous release of the operating system.
- You cannot use the TGTRLS parameter for COBOL programs created in the System/38 environment.
- You can restore an object program to the previous release or to a subsequent release. You cannot restore an object program on a system that is more than one release lower. That is, only one release of downward compatibility is provided.
- You cannot use functions that are new to the current release of the operating system in a program that you compile for use at the previous release level.
- Programs may be retranslated when they are restored to the previous release; therefore, you cannot delete observability if the programs are to be retranslated.
- No product library should be in the system portion of your library list.
Using the PROCESS Statement to Specify Compiler Options

The PROCESS statement is an optional part of the COBOL source program. You can use the PROCESS statement to specify options you would normally specify at compilation time. Options specified in the PROCESS statement override the corresponding options specified in the CRTCBPGM CL command.

The format of the PROCESS statement is as follows:

```
Format

/PROCESS option-1
```

The following rules apply:

- The statement must be placed before the first source statement in the COBOL program immediately preceding the IDENTIFICATION DIVISION header.
- The statement begins with the word PROCESS. Options can appear on more than one line; however, only the first line can contain the word PROCESS.
- The word PROCESS and all options must appear within positions 8 through 72. Position 7 must be left blank. The remaining positions can be used as in COBOL source statements: positions 1 through 6 for sequence numbers, positions 73 through 80 for identification purposes.
- The options must be separated by blanks and/or commas.
- Options can appear in any order. If conflicting options are specified, for example, LIST and NOLIST, the last option encountered takes precedence.
- If the option keyword is correct and the suboption is in error, the default suboption is assumed.

Not every CRTCBPGM parameter has a corresponding option in the PROCESS statement. Refer to the following tables which indicate the allowable PROCESS statement options and the equivalent CRTCBPGM command parameters and options. Defaults are underlined. Descriptions of the PROCESS statement options correspond to the parameter and option descriptions that start on page 18.

<table>
<thead>
<tr>
<th>PROCESS Statement Option</th>
<th>CRTCBPGM</th>
</tr>
</thead>
<tbody>
<tr>
<td>GENLVL(nn)</td>
<td>GENLVL Parameter Option</td>
</tr>
<tr>
<td></td>
<td>nn</td>
</tr>
<tr>
<td>PROCESS Statement Options</td>
<td>CRTCBLPGM</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td><strong>GEN</strong></td>
<td>*GEN</td>
</tr>
<tr>
<td><strong>NOGEN</strong></td>
<td>*NOGEN</td>
</tr>
<tr>
<td><strong>NOMAP</strong></td>
<td>*NOMAP</td>
</tr>
<tr>
<td><strong>MAP</strong></td>
<td>*MAP</td>
</tr>
<tr>
<td><strong>NONUMBER</strong></td>
<td>*NONUMBER</td>
</tr>
<tr>
<td><strong>NUMBER</strong></td>
<td>*NUMBER</td>
</tr>
<tr>
<td><strong>LINENUMBER</strong></td>
<td>*LINENUMBER</td>
</tr>
<tr>
<td><strong>NOSECVL</strong></td>
<td>*NOSECVL</td>
</tr>
<tr>
<td><strong>SECVL</strong></td>
<td>*SECVL</td>
</tr>
<tr>
<td><strong>NOOPTIONS</strong></td>
<td>*NOOPTIONS</td>
</tr>
<tr>
<td><strong>OPTIONS</strong></td>
<td>*OPTIONS</td>
</tr>
<tr>
<td><strong>QUOTE</strong></td>
<td>*QUOTE</td>
</tr>
<tr>
<td><strong>APOST</strong></td>
<td>*APOST</td>
</tr>
<tr>
<td><strong>NOSEQUENCE</strong></td>
<td>*NOSEQUENCE</td>
</tr>
<tr>
<td><strong>SEQUENCE</strong></td>
<td>*SEQUENCE</td>
</tr>
<tr>
<td><strong>SOURCE</strong> (or SRC)</td>
<td>*SOURCE (or *SRC)</td>
</tr>
<tr>
<td><strong>NOSOURCE</strong> (or NOSRC)</td>
<td>*NOSOURCE (or *NOSRC)</td>
</tr>
<tr>
<td><strong>NOVBSUM</strong></td>
<td>*NOVBSUM</td>
</tr>
<tr>
<td><strong>VBSUM</strong></td>
<td>*VBSUM</td>
</tr>
<tr>
<td><strong>NOXREF</strong></td>
<td>*NOXREF</td>
</tr>
<tr>
<td><strong>XREF</strong></td>
<td>*XREF</td>
</tr>
<tr>
<td><strong>PRTCORR</strong></td>
<td>*PRTCORR</td>
</tr>
<tr>
<td><strong>NOPRTCORR</strong></td>
<td>*NOPRTCORR</td>
</tr>
</tbody>
</table>

**OPTION Parameter Options**

Chapter 3. Compiling a Program 33
### PROCESS Statement Options

<table>
<thead>
<tr>
<th>PROCESS Statement Options</th>
<th>CRTCBBLPGM</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOINZDLT</td>
<td>*NOINZDLT</td>
</tr>
<tr>
<td>INZDLT</td>
<td>*INZDLT</td>
</tr>
<tr>
<td>NOLIST</td>
<td>*NOLIST</td>
</tr>
<tr>
<td>LIST</td>
<td>*LIST</td>
</tr>
<tr>
<td>STDERR</td>
<td>*STDERR</td>
</tr>
<tr>
<td>NOSTDERR</td>
<td>*NOSTDERR</td>
</tr>
<tr>
<td>NODDSFILLER</td>
<td>*NODDSFILLER</td>
</tr>
<tr>
<td>DDSFILLER</td>
<td>*DDSFILLER</td>
</tr>
<tr>
<td>NOSYNC</td>
<td>*NOSYNC</td>
</tr>
<tr>
<td>SYNC</td>
<td>*SYNC</td>
</tr>
<tr>
<td>NOCRTF</td>
<td>*NOCRTF</td>
</tr>
<tr>
<td>CRTF</td>
<td>*CRTF</td>
</tr>
<tr>
<td>NODUPKEYCHK</td>
<td>*NODUPKEYCHK</td>
</tr>
<tr>
<td>DUPKEYCHK</td>
<td>*DUPKEYCHK</td>
</tr>
<tr>
<td>NOEXTACCDSP</td>
<td>*NOEXTACCDSP</td>
</tr>
<tr>
<td>EXTACCDSP</td>
<td>*EXTACCDSP</td>
</tr>
<tr>
<td>NOBLK</td>
<td>*NOBLK</td>
</tr>
<tr>
<td>BLK</td>
<td>*BLK</td>
</tr>
<tr>
<td>STDINZ</td>
<td>*STDINZ</td>
</tr>
<tr>
<td>NOSTDINZ</td>
<td>*NOSTDINZ</td>
</tr>
<tr>
<td>FS21DUPKEY</td>
<td>*FS21DUPKEY</td>
</tr>
<tr>
<td>NOFS21DUPKEY</td>
<td>*NOFS21DUPKEY</td>
</tr>
<tr>
<td>RANGE</td>
<td>*RANGE</td>
</tr>
<tr>
<td>NORANGE</td>
<td>*NORANGE</td>
</tr>
<tr>
<td>UNREF</td>
<td>*UNREF</td>
</tr>
<tr>
<td>NOUNREF</td>
<td>*NOUNREF</td>
</tr>
</tbody>
</table>

### CVTOPT Parameter Options

<table>
<thead>
<tr>
<th>PROCESS Statement Options</th>
<th>CRTCBBLPGM</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOVARCHAR</td>
<td>*NOVARCHAR</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>*VARCHAR</td>
</tr>
<tr>
<td>NODATETIME</td>
<td>*NODATETIME</td>
</tr>
<tr>
<td>DATETIME</td>
<td>*DATETIME</td>
</tr>
<tr>
<td>NOCVTGRAPHIC</td>
<td>*NOGRAPHIC</td>
</tr>
<tr>
<td>CVTGRAPHIC</td>
<td>*GRAPHIC</td>
</tr>
<tr>
<td>PROCESS Statement Options</td>
<td>CRTCBLPGM</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>NOFIPS</td>
<td>*NOFIPS</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>*MINIMUM</td>
</tr>
<tr>
<td>INTERMEDIATE</td>
<td>*INTERMEDIATE</td>
</tr>
<tr>
<td>HIGH</td>
<td>*HIGH</td>
</tr>
<tr>
<td>NOSEG</td>
<td>*NOSEG</td>
</tr>
<tr>
<td>SEG1</td>
<td>*SEG1</td>
</tr>
<tr>
<td>SEG2</td>
<td>*SEG2</td>
</tr>
<tr>
<td>NODEB</td>
<td>*NODEB</td>
</tr>
<tr>
<td>DEB1</td>
<td>*DEB1</td>
</tr>
<tr>
<td>DEB2</td>
<td>*DEB2</td>
</tr>
<tr>
<td>NOOBSOLETE</td>
<td>*NOOBSOLETE</td>
</tr>
<tr>
<td>OBSOLETE</td>
<td>*OBSOLETE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PROCESS Statement Options</th>
<th>CRTCBLPGM</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTDSPOPT(a b c)</td>
<td>EXTDSPOPT Parameter Options</td>
</tr>
<tr>
<td>DFRWRT</td>
<td>*DFRWRT</td>
</tr>
<tr>
<td>NODFRWRT</td>
<td>*NODFRWRT</td>
</tr>
<tr>
<td>UNDSPCHR</td>
<td>*UNDSPCHR</td>
</tr>
<tr>
<td>NOUNDSPCHR</td>
<td>*NOUNDSPCHR</td>
</tr>
<tr>
<td>ACCUPDALL</td>
<td>*ACCUPDALL</td>
</tr>
<tr>
<td>ACCUPDNE</td>
<td>*ACCUPDNE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PROCESS Statement Options</th>
<th>CRTCBLPGM</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOSAAFLAG</td>
<td>*NOFLAG</td>
</tr>
<tr>
<td>SAAFLAG</td>
<td>*FLAG</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PROCESS Statement Option</th>
<th>CRTCBLPGM</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLAG(nn)</td>
<td>nn</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PROCESS Statement Options</th>
<th>CRTCBLPGM</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOFS9MTO0M</td>
<td>not applicable</td>
</tr>
<tr>
<td>FS9MTO0M</td>
<td>not applicable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PROCESS Statement Options</th>
<th>CRTCBLPGM</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOGRAPHIC</td>
<td>not applicable</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>not applicable</td>
</tr>
</tbody>
</table>

FS9MTO0M changes a file status of 9M to a file status of 0M.
The GRAPHIC option of the PROCESS statement is available for processing DBCS characters in DBCS literals. See Appendix F, “Supporting International Languages with Double-Byte Character Sets” on page 337 for information about DBCS support.

The EXTDSPOPT option on the PROCESS statement should be coded with the associated options in brackets similar to FLAG(nn) syntax. You can specify more than one option within the brackets for the EXTDSPOPT option. For example, to specify DFRWRT and UNDSPCHR, type

```
EXTDSPOPT(DFRWRT UNDSPCHR)
```

It is also valid to specify EXTDSPOPT or EXTDSPOPT().

When EXTDSPOPT alone is specified in the PROCESS statement, then all the default values for the additional options are in effect.

If you specify EXTDSPOPT(), it has no effect on your program.

If conflicting options are specified, the last option specified overrides the others.

**Compiling Multiple Programs**

The PROCESS statement can be used to separate multiple programs and/or sub-programs to be compiled with a single invocation of the compiler. (A subprogram is a called program that is combined with the calling program at run time to produce a run unit.) When compiling multiple programs, all compiler options specified on the CRTCBPGM command statement, plus all default options, plus the options specified on the last PROCESS statement preceding the program will be in effect for the compilation of that program. All compiler output is directed to the destinations specified by the CRTCBPGM command statement.

All object programs are stored in the library specified on the PGM parameter. If program-name is specified for the PGM parameter, the first program in the batch job has that name, and all other programs use the name specified in the PROGRAM-ID paragraph in the source program.

**Using COPY within the PROCESS Statement**

A COPY statement can be used in the source program wherever a character-string or separator can be used. Each COPY statement must be preceded by a space and followed by a period and a space. For more information on the COPY statement, refer to the “COPY Statement” section of the COBOL/400 Reference.

The Format 1 COPY statement can be used within the PROCESS statement to retrieve compiler options previously stored in a source library, and include them in the PROCESS statement. COPY can be used to include options that override those specified as defaults by the compiler. Any PROCESS statement options can be retrieved with the COPY statement.

Compiler options can both precede and follow the COPY statement within the PROCESS statement. The last encountered occurrence of an option overrides all preceding occurrences of that option.
The following example shows the use of the COPY statement within the PROCESS statement. The COPY statement must be followed by a period. Notice also that, in this example, NOMAP overrides the corresponding option in the library member:

```
000001  PROCESS XREF       MYPROG
000002  COPY DFLTS.        MYPROG
          MAP, SOURCE, LIST     DFLTS
000004  NOMAP, FLAG(20)    MYPROG
000010  IDENTIFICATION DIVISION.  MYPROG
```

### Understanding Compiler Output

Compiler output can include:

- A summary of command options
- An options listing: a listing of options in effect for the compilation. Use OPTION(*OPTIONS).
- A source listing: a listing of the statements contained in the source program. Use OPTION(*SOURCE or *SRC).
- A verb usage listing: a listing of the COBOL verbs and the number of times each verb is used. Use OPTION(*VBSUM).
- A Data Division map: a glossary of compiler-generated information about the data. Use OPTION(*MAP). Also included is a mapping of user-supplied names to compiler-generated internal names.
- SAA flagging: a list of the functions in your program that are not portable to other SAA COBOL environments. Use SAAFLAG(*FLAG).
- FIPS messages: a list of messages for a FIPS COBOL subset, for any of the optional modules, for all of the obsolete language elements, or for a combination of a FIPS COBOL subset, optional modules and all obsolete elements. Refer to the information on the “FLAGSTD Parameter” on page 25 for the specific options available for FIPS flagging.
- A cross-reference listing. Use OPTION(*XREF).
- Compiler messages (including diagnostic statistics).
- Compilation statistics.
- A listing of the generated program in symbolic form.
- An object program.

The presence or absence of some of these types of compiler output is determined by options specified in the PROCESS statement or through the CRTCBLPGM command. The level of diagnostic messages printed depends upon the FLAG option.
Specifying the Format of Your Listing

A slash (/) in the indicator area (column 7) of a line results in page ejection of the source program listing. The slash (/) comment line prints on the first line of the next page.

If you specify the EJECT statement in your program, the next source statement prints at the top of the next page in the compiler listing. This statement may be written anywhere in Area A or Area B and must be the only statement on the line.

The SKIP1/2/3 statement causes blank lines to be inserted in the compiler listing. A SKIP1/2/3 statement can be written anywhere in Area A or B. It must be the only statement on the line.

- SKIP1 inserts a single blank line (double spacing).
- SKIP2 inserts two blank lines (triple spacing).
- SKIP3 inserts three blank lines (quadruple spacing).

Each of the above SKIP statements causes a single insertion of one, two, or three lines.

A TITLE statement places a title on each indicated page.

You can selectively list or suppress your COBOL source statements by using the *CONTROL, *CBL, or COPY statements:

- *CONTROL NOSOURCE and *CBL NOSOURCE suppress the listing of source statements.
- *CONTROL SOURCE and *CBL SOURCE continue the listing of source statements.
- A COPY statement bearing the SUPPRESS phrase suppresses the listing of copied statements. For its duration, this statement overrides any *CONTROL or *CBL statement. If the copied member contains *CONTROL or *CBL statements, the last one runs once the COPY member has been processed.

Refer to the COBOL/400 Reference for additional information about the EJECT, SKIP1/2/3, *CONTROL, *CBL, COPY, and TITLE statements.

Time-Separation Characters

The TIMSEP parameter of job-related commands (such as CHGJOB) now specifies the time-separation character used in the time stamps that appear on compiler listings. In the absence of a TIMSEP value, the system value QTIMSEP is used by default.
Browsing Your Compiler Listing Using SEU

The Source Entry Utility (SEU) allows you to browse through a compiler listing in an output queue. You can review the results of a previous compilation while making the required changes to your source code. Figure 5 shows the split-display in SEU that allows you to browse through the listing from a work station.

```
Columns . . . : 1 71  Edit  XMPLIB/QLBLSRC
SEU=>XMPLE
FMT CB ------A+++B+---------------------------------------------
0014.00    DATA DIVISION.
0015.00    FILE SECTION.
0016.00    FD FILE1
0017.00    RECORD CONTAINS 56 CHARACTERS
0018.00    LABEL RECORDS ARE OMITTED
0019.00    DATA RECORD IS REB-1.
0020.00    01 REC-1 PIC X(56).

Columns . . . : 1 71  Browse  Spool file . . : XMＰLE
SEU=>
0000.50    STMT
0000.51    * 19 MSGID: LBL1327 SEVERITY: 30 SEQNBR: 001900
0000.52    Message . . . . : 'REB-1' not defined in the program. Clause ignored.
0000.54    *** END OF MESSAGES ***
0000.55    Message Summary
0000.56    Total Info(0-4) Warning(5-19) Error(20-29) Severe(30-39)
F6=Move split line  F19=Left  F20=Right
F21=System command  F24=More keys
Syntax error found.
```

Figure 5. SEU Split Edit/Browse Display

While browsing the compiler listing, you can scan for errors and correct those source statements that have errors. To scan for errors, type F ERR on the SEU command line.

For complete information on browsing through a compiler listing, see the SEU User's Guide and Reference.

A Sample Program and Listing

The following pages illustrate the compiler options and source listing produced for the program example. References to the figures are made throughout the following text. These references are indexed by the reverse printing of letters on a black background, for example ( Z ). The reverse letters in the text correspond to the letters found in the figures.

Command Summary

This summary, which is produced as a result of compilation, lists all options specified in the CRTCBLPGM command. Refer to "Using the Create COBOL Program (CRTCBLPGM) Command" on page 15 for more information about user-defined options.
Identifying the Compiler Options in Effect

The PROCESS statement, if specified, is printed first. Figure 7 is a list of all options in effect for the compilation of the program example: the options specified in the CRTCBLPGM command, as modified by the PROCESS statement. Compiler options are listed at the beginning of all compiler output when the OPTIONS parameter is specified.
Source Listing

Figure 8 illustrates a source listing. The statements in the source program are listed exactly as submitted. The source is not listed if the NOSOURCE option is specified. After the page in which the PROGRAM-ID paragraph is listed, all compiler output pages have the program-id name listed in the heading before the system name.
Figure 8 (Part 1 of 2). An Example of a COBOL/400 Source Listing
Figure 8 (Part 2 of 2). An Example of a COBOL/400 Source Listing

Figure 8 displays the following fields:

A. **Compiler-generated statement number**: The numbers appear to the left of the source program listing. These numbers are referenced in all compiler output listings except for FIPS messages listings. A statement number can span several lines, and a line can contain more than one statement.

B. **Reference number**: The numbers appear to the left of the source statements. The numbers that appear in this field and the column heading (shown as SEQNBR in this listing) are determined by an option specified in the CRTCBPGM command or in the PROCESS statement, as shown in the following table:

<table>
<thead>
<tr>
<th>Option</th>
<th>Heading</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONUMBER</td>
<td>SEQNBR</td>
<td>Source-file sequence numbers</td>
</tr>
<tr>
<td>NUMBER</td>
<td>NUMBER</td>
<td>Standard COBOL sequence numbers</td>
</tr>
<tr>
<td>LINENUMBER</td>
<td>LINNBR</td>
<td>Compiler-generated sequence numbers</td>
</tr>
</tbody>
</table>

C. **Sequence error indicator column**: An S in this column indicates that the line is out of sequence. Sequence checking is performed on the reference number field only if the SEQUENCE option is specified.

D. **Copyname**: The copyname, as specified in the COBOL COPY statement, is listed here for all records included in the source program by that COPY statement. If the DDS-ALL-FORMATS phrase is used, the name <--ALL-FMTS appears under COPYNAME.

E. **Change/date field**: The date the line was last modified is listed here.

**Verb Usage by Count Listing**

Figure 9 shows the alphabetic list that is produced of all verbs used in the source program. A count of how many times each verb was used is also included. This listing is produced when the VBSUM option is specified.
Figure 9. Verb Usage by Count Listing

**Data Division Map**

The Data Division map is listed when the MAP option is specified. It contains information about names in the COBOL source program. The number of bytes required for the File Section and Working-Storage Section is given at the end of the Data Division map.

The Data Division map displays the following fields:

- **Statement number**: The compiler-generated statement number where the data item was defined is listed for each data item in the Data Division map.

- **Level of data item**: The level number of the data item, as specified in the source program, is listed here. Index-names are identified by an IX in the level-number and a blank type field.

- **Source name**: The data name, as specified in the source program, is listed here.
**Section:** The section where the item was defined is shown here through the use of the following codes:

- **FS** File Section
- **WS** Working-Storage Section
- **LS** Linkage Section
- **SM** Sort/Merge Section
- **SR** Special Register.

**Displacement:** The offset, in bytes, of the item from the level-01 group item is given here.

**Length:** The decimal length of the item is listed here.

**Type:** The data class type for the item is shown here through the use of the following codes:

<table>
<thead>
<tr>
<th>GROUP</th>
<th>Group Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Alphabetic</td>
</tr>
<tr>
<td>AN</td>
<td>Alphanumeric</td>
</tr>
<tr>
<td>ANE</td>
<td>Alphanumeric edited</td>
</tr>
<tr>
<td>INDEX</td>
<td>Index data item (USAGE INDEX)</td>
</tr>
<tr>
<td>BOOLN</td>
<td>Boolean</td>
</tr>
<tr>
<td>ZONED</td>
<td>Zoned decimal (external decimal)</td>
</tr>
<tr>
<td>PACKED</td>
<td>Packed decimal (internal decimal) (USAGE COMP, COMP-3 or PACKED-DECIMAL)</td>
</tr>
<tr>
<td>BINARY</td>
<td>Binary (USAGE COMP-4 or BINARY)</td>
</tr>
<tr>
<td>NE</td>
<td>Numeric edited</td>
</tr>
<tr>
<td>POINTR</td>
<td>Pointer data item (USAGE POINTER)</td>
</tr>
</tbody>
</table>

**Internal name:** The compiler-generated internal names are listed here and are assigned as follows:

**File names**

The internal name uses the form `.Fnn`, where `.F` indicates a file name, and `nn` is a unique two-digit number.

**Data names**

The internal name uses the form `.Dxxxxxx`, where `.D` indicates a data name or index name, and `xxxxxx` is a unique six-digit hex value. These names appear in the IRP listing if generated.

**Attributes:** The attributes of the item are listed here as follows:

- For files, the following information can be given:
  
  Device type
  ORGANIZATION
  ACCESS MODE
  BLOCK CONTAINS information
  RECORD CONTAINS information
  LABEL information
  RERUN is indicated
  SAME AREA is indicated
  CODE-SET is indicated
  SAME RECORD AREA is indicated
  LINAGE is indicated.
• For data items, the attributes indicate if the following information was specified for the item:
  
  REDEFINES
  VALUE
  JUSTIFIED
  SYNCHRONIZED
  BLANK WHEN ZERO
  SIGN IS LEADING
  SIGN IS LEADING SEPARATE
  SIGN IS SEPARATE
  INDICATORS.

• For table items, the dimensions for the item are listed here in the form DIMENSION (nn). For each dimension, a maximum OCCURS value is given. When a dimension is a variable, it is listed as such, giving the lowest and highest OCCURS values.

FIPS Messages

The FIPS messages, Figure 11, are listed when the FLAGSTD parameter is specified. See page 25 for more information about specifying the option for FIPS flagging. Only messages for the requested FIPS subset, optional modules and/or obsolete elements are listed.

Note: The sequence number and column number are given for each time the message is issued.

Figure 11. FIPS Messages

The FIPS messages consist of the following fields:

- **FIPS-ID**: This field lists the FIPS message number.
- **Description and reference numbers flagged**: This field lists a description of the condition flagged, followed by a list of the reference numbers from the source program where this condition is found.
The type of reference numbers used, and their names in the heading (shown as SEQUENCE NUMBERS in this listing) are determined by an option specified in the CRTCBLPGM command or in the PROCESS statement, as shown in the following table:

<table>
<thead>
<tr>
<th>Option</th>
<th>Heading</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONUMBER</td>
<td>DESCRIPTION AND SEQUENCE NUMBERS FLAGGED</td>
</tr>
<tr>
<td>NUMBER</td>
<td>DESCRIPTION AND USER-SUPPLIED NUMBERS FLAGGED</td>
</tr>
<tr>
<td>LINENUMBER</td>
<td>DESCRIPTION AND LINE NUMBERS FLAGGED</td>
</tr>
</tbody>
</table>

- **Items grouped by level**: These headings subdivide the FIPS messages by level and category.
- **FIPS violations flagged**: The total number of FIPS violations flagged is included at the end of the FIPS listing.

### SAA Messages

Figure 12 shows the SAA messages that are listed when you specify the SAA flagging option. See the SAAFLAG parameter on page 25 or “Using the PROCESS Statement to Specify Compiler Options” on page 32 for more information about specifying this option.

Figure 12. SAA Messages

![SAA COBOL Messages](image)

For more information about SAA flagging, see “SAA Flagging” on page 333.

### Cross-Reference Listing

Figure 13 shows the cross-reference listing, which is produced when the XREF option is specified. It provides a list of all data references and procedure-name references, by statement number, within the source program.
The cross-reference listing displays the following fields:

**Names field:** The data name or procedure name referenced is listed here. All procedure names are flagged with an * before the name. The names are listed alphabetically.

**Defined field:** The statement number where the name was defined within the source program is listed here.

**References field:** All statement numbers are listed in the same sequence as the name is referenced in the source program. An * following a statement number indicates that the item was modified in that statement.

**Messages**

Figure 14 shows the messages that are generated during program compilation.

---

**Figure 14. Diagnostic Messages**

- **Source records read**: 79
- **Copy records read**: 10
- **Copy members processed**: 1
- **Sequence errors**: 0
- **Highest severity message issued**: 0

LBL0801 DD Program SAMPLE created in library TESTER.

---

**Figure 13. Cross-Reference Listing**

The cross-reference listing displays the following fields:

**Names field:** The data name or procedure name referenced is listed here. All procedure names are flagged with an * before the name. The names are listed alphabetically.

**Defined field:** The statement number where the name was defined within the source program is listed here.

**References field:** All statement numbers are listed in the same sequence as the name is referenced in the source program. An * following a statement number indicates that the item was modified in that statement.
The fields displayed are:

**Statement number:** This field lists the compiler-generated statement number associated with the statement in the source program for which the message was issued.\(^1\)

**Reference number:** The reference number is issued here.\(^1\) The numbers that appear in this field and the column heading (shown here as SEQNBR) are determined by an option specified in the CRTCBLPGM command or in the PROCESS statement, as shown in the following table:

<table>
<thead>
<tr>
<th>Option</th>
<th>Heading</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONUMBER</td>
<td>SEQNBR</td>
<td>Source-file sequence numbers</td>
</tr>
<tr>
<td>NUMBER</td>
<td>NUMBER</td>
<td>User-supplied sequence numbers</td>
</tr>
<tr>
<td>LINENUMBER</td>
<td>LINNBR</td>
<td>Compiler-generated sequence numbers</td>
</tr>
</tbody>
</table>

When a message is issued for a record from a copy file, the number is preceded by a +.

**MSGID and Severity Level:** These fields contain the message number and its associated severity level. Severity levels are defined as follows:

- 00 Informational
- 10 Warning
- 20 Error
- 30 Severe Error
- 40 Unrecoverable (usually a user error)
- 50 Unrecoverable (usually a compiler error)

**Message:** The message identifies the condition and indicates the action taken by the compiler.

**Message statistics:** This field lists the total number of messages and the number of messages by severity level.

The totals listed are the number of messages generated for each severity by the compiler and are not always the number listed. For example, if FLAG(10) is specified, no messages of severity less than 10 are listed. The counts, however, do indicate the number that would have been printed if they had not been suppressed.

---

\(^1\) The statement number and the reference number do not appear on certain messages that reference missing items. For example, if the PROGRAM-ID paragraph is missing, message LBL0031 appears on the listing with no statement or reference number listed.

Chapter 3. Compiling a Program 49
Chapter 4. Running Your COBOL Program

This chapter provides the information you need to run your COBOL/400 program.

The most common ways to run a COBOL program are:

- Using a Control Language (CL) CALL command
- Using the COBOL CALL statement
- Using a menu-driven application program
- Issuing a user-created command.

You can use a CL CALL command interactively, as part of a batch job, or include it in a CL program. An example of a CL CALL command is CALL PAYROLL. PAYROLL is the name of a COBOL program that is called and run.

Any COBOL program can call another program with the COBOL CALL statement. (See the “CALL Statement” section of the COBOL/400 Reference for more information.)

Another way to run a COBOL program is from a menu-driven application. The workstation user selects an option from a menu, calling the appropriate program. The following figure illustrates an example of an application menu.

```
PAYROLL DEPARTMENT MENU

1. Inquire into employee master
2. Change employee master
3. Add new employee
4. Return

Option:____
```

Figure 15. Example of an Application Menu

The menu shown in this figure is normally displayed by a CL program in which each option calls a separate COBOL program.

You can also create a command yourself to run a COBOL program by using a command definition. A command definition is an object that contains the definition of a command (including the command name, parameter descriptions, and validity-checking information), and identifies the program that performs the function requested by the command. The system-recognized identifier for the object is *CMD.

For example, you can create a command, PAY, that calls a program, PAYROLL. PAYROLL is the name of a COBOL program that is called and run. You can enter the command interactively, or in a batch job. See the CL Programmer’s Guide for further information about using the command definition.

When a COBOL program ends normally, the system returns control to the caller. The caller could be a workstation user, a CL program (such as the menu-handling program), or another COBOL program.
If a COBOL program ends abnormally during run time, the escape message LBE9001

Error message-id caused program to end.

is issued. A CL program can monitor for this exception by using the Monitor Message (MONMSG) command. See the CL Reference for more information about control language commands.

If a program ends for any reason other than by the use of the STOP statement or by falling through to the end of the program, the return code is set to 2. See the RTVJOBA and DSPJOB commands in the CL Programmer’s Guide for more information about return codes.

When you are running a batch job that uses the ACCEPT statement, the input data is taken from the job stream. This data must be placed immediately following the CL CALL for the COBOL program. It is your responsibility to request (through multiple ACCEPT statements) the same amount of data as is available. See the “ACCEPT Statement” section of the COBOL/400 Reference for more information.

Note: If more data is requested than is available, the CL command following the data is treated as input data. If more data is available than is requested, each extra line of data is treated as a CL command. In each instance, undesirable results can occur.

Replying to Run-Time Inquiry Messages

When you run a COBOL program, run-time inquiry messages may be generated. The messages require a response before the program continues running.

You can add the inquiry messages to a system reply list to provide automatic replies to the messages. The replies for these messages may be specified individually or generally. This method of replying to inquiry messages is especially suitable for batch programs, which would otherwise require an operator to issue replies.

You can add the following COBOL/400 inquiry messages to the system reply list:

LBE7200
LBE7201
LBE7203
LBE7204
LBE7205
LBE7206
LBE7207
LBE7208
LBE7209
LBE7210
LBE7211
LBE7604.

The reply list is only used when an inquiry message is sent by a job that has the Inquiry Message Reply (INQMSGRPY) attribute specified as INQMSGRPY(*SYSRPYL).

The INQMSGRPY parameter occurs on the following CL commands:

- Change Job (CHGJOB)
- Change Job Description (CHGJOBD)
• Create Job Description (CRTJOB)
• Submit Job (SBMJOB).

You can select one of four reply modes by specifying one of the following values for the INQMSGRPY parameter:

SAME No change is made in the way that replies are sent to inquiry messages
RQD All inquiry messages require a reply by the receiver of the inquiry messages
DFT A default reply is issued
SYSRPYL The system reply list is checked for a matching reply list entry. If a match occurs, the reply value in that entry is used. If no entry exists for that inquiry message, a reply is required.

You can use the Add Reply List Entry (ADDRPYLE) command to add entries to the system reply list, or the Work with Reply List Entry (WRKRPYLE) command to change or remove entries in the system reply list. See the CL Reference for details of the ADDRPYLE and WRKRPYLE commands. You can also reply to runtime inquiry messages with a user-defined error-handler. For more information about error-handling APIs, refer to the System Programmer’s Interface Reference.
Chapter 5. Debugging Your Program

The COBOL/400 language and the OS/400 operating system provide functions for debugging the programs you develop. This chapter describes those functions that allow you to debug your programs.

<table>
<thead>
<tr>
<th>OS/400 Functions</th>
<th>COBOL/400 Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakpoints</td>
<td>Debugging features</td>
</tr>
<tr>
<td>Traces</td>
<td>Formatted dump</td>
</tr>
</tbody>
</table>

The OS/400 functions let you test programs while protecting your production files, and let you observe and debug operations as a program runs. No special source code is required for using the OS/400 functions.

The COBOL functions can be used independently of the OS/400 functions or in combination with them to:

- Debug a program
- Produce a formatted dump of the contents of fields, data structures, arrays, and tables.

Source code is required for using COBOL debugging features and formatted dump capability. A formatted dump can also be obtained by a user’s response to a runtime message.

OPEN-FEEDBACK and I-O-FEEDBACK contents can provide additional debugging information. The method for obtaining this information is described later in this chapter in “File Status and Feedback Areas” on page 103.

While testing your programs, ensure that your library list is changed to direct the programs to a test library containing test data so that any existing real data is not affected.

To prevent database files in production libraries from being modified unintentionally, you can specify UPDPROD(*NO) on the Start Debug (STRDBG) command or by using the Change Debug (CHGDBG) command. See the CL Reference for more information.

Note: Refer to the CL Programmer’s Guide for the CL commands required for testing and debugging programs.

No special statements for testing are contained in the program being tested. The program can be run normally without modification. All testing functions are specified in the job that contains the program, not in the actual program.

Testing functions apply only to the job in which they are specified. A program can be used concurrently in two jobs: one job that is in a test environment and another that is in a normal processing environment.

Testing functions allow you to observe the operations being performed while the program is running. These functions include using breakpoints and traces. (See “Using Breakpoints” on page 57 and “Using a Trace” on page 64 for more information.)
Avoiding Common Coding Errors

The errors made most frequently by COBOL programmers fall into two classes: compilation-time errors and run-time errors.

The compiler can detect errors when compiling your source program. While it makes corrections based on assumptions about certain errors it finds, you still need to correct the source and compile again if you have errors.

Common coding mistakes include:

- Unmatched record descriptions with externally described files
- Missing copy files
- Misspellings
- Faulty punctuation, especially missing periods
- Incorrect or incomplete syntax
- Misuse of reserved words.

The following errors appear only when you run your program:

- Failing to match the record description in your source program with the format of the actual records on the file to be read. This can either be an error by you (the records are correct, but your description is incorrect) or an error by the person who created the records your program reads. (For example, your description is correct, but one or more records were entered incorrectly.)

- Moving a data item whose subscript or index is too large, is negative, or is 0. Such a move could overlay and destroy part of your code or could fetch faulty data.

- Forgetting to define a sign field for items that can hold negative values. (In such a case, the sign is lost, and the negative number mistakenly becomes positive.)

- Moving data into an area too small for it, causing unwanted truncation.

- Forgetting to initialize the data items in the Working-Storage section before they are used. This may result in a decimal data error.

- In a called program, incorrectly matching the data descriptions in the Linkage Section with those of the caller. Or, in the calling program, incorrectly identifying the data to be passed.

- Moving a group item to another group item when the subordinate data descriptions are incompatible.

- Specifying USAGE for a redefined data item that is different from the USAGE originally specified for the redefined item, and then forgetting about the change once the redefinition takes place.

- Including a GO TO statement with no procedure name, and failing to initialize it with an ALTER statement before the running program reaches that point.

- Failing to include the AT END or INVALID KEY clauses or the USE procedures on files described in the program.

- Failing to match the TRANSACTION file source record description with the display format record description.
Using Breakpoints

A breakpoint is a statement number or a label in your program that stops program processing, and gives control to the display station user or to a specified program. If you use a statement number, it can be a statement number that appears on the compiler listing of the COBOL source program. If you use a label as a breakpoint, the label can be:

- Associated with a function performed by your COBOL program. For example, .OPEN indicates the open file function.
- An internal COBOL compiler generated label. For example, .L000001 indicates the first internally generated label.

**Note:** To determine the internally generated labels for your program, use the GENOPT parameter on the CRTCBBLPGM command to get an IRP listing of the program.

When a breakpoint statement is about to be run for an interactive job, the system displays the breakpoint at which the program has stopped and, if requested, the values of program variables. After you get this information (in a display), you can go to a Command Entry display and then enter OS/400 commands to request other functions (such as displaying or changing a variable, adding a breakpoint, or adding a trace). See the *CL Programmer’s Guide* for more information on breakpoint concepts.

For a batch job, a breakpoint program can be called when a breakpoint is reached. The breakpoint information is passed to the breakpoint program.

**Example of a Program Using Breakpoints**

Figure 16 shows an example of a COBOL program using breakpoints. The following OS/400 commands add breakpoints at statements 43 and 52. The value of variable KOUNT is displayed when the breakpoint at statement 52 is reached.

OS/400 Commands:

```
STRDBG TESTPRT
ADDBKP STMT(43)
ADDBKP STMT(52)
PGMVAR(KOUNT)
```

The OS/400 commands are explained in the *CL Reference*. 
Figure 16 (Part 1 of 2). Example of a COBOL Program Using Breakpoints
The first breakpoint shows you where you are in the program. The following information is displayed when the break occurs:

Display Breakpoint

Statement/Instruction . . . . . . . . : 43 /0017
Program . . . . . . . . . . . . . . . . : TESTPRT
Recursion level . . . . . . . . . . . : 1

Press Enter to continue.

F3=Exit Program  F10=Command entry

Figure 17. First Breakpoint Displayed
The following information is displayed as a result of reaching the second breakpoint:

![Display Breakpoint](image)

Press Enter to continue.
F3=Exit Program  F10=Command entry

---

**Figure 18. Second Breakpoint Displayed**

To specify a variable for the PGMVAR parameter, begin every name you enter with an alphanumeric character (A through Z, $, #, or @). It can be followed by the characters (A through Z, 0 though 9, $, #, @, or _).

The following example shows how to display a COBOL variable, RECORD-NO, in the program example. Because the hyphen is treated by the OS/400 operating system as a special character, RECORD-NO must be enclosed in quotation marks.

```
PGMVAR('RECORD-NO')
```

To display the value of a table element, enter the appropriate occurrence numbers (subscripts) with the variable name. Up to seven dimensions of subscripting are allowed, and the subscripts must be separated by commas.

Do not use an index-name or index data-item as a subscript. When an index is entered as a subscript, the operating system uses the internal value of the index as the subscript, and undesirable results can occur.

The following example shows how to specify the COBOL variable TABLE1 with three dimensions.

```
PGMVAR('TABLE1(SUB1, SUB2, SUB3)')
```

One or more blanks are allowed after each comma separating subscripts, but the total length of the variable plus subscripts, parentheses, commas, and blanks specified with the PGMVAR keyword cannot exceed 132 characters. For more information on how to code variables in CL commands, see the *CL Reference*. 
Variable names can be qualified in the PGMVAR parameter. For example:

```
PGMVAR('NAME-FIELD OF WORK-RECORD')
```

Another technique can be used to display variables that are not elements of a multi-dimensional table. For example, to display the field NAME-FIELD, you can use the COBOL Data Division map to find its COBOL internal name (I-NAME). Next, use the IRP cross-reference listing to find the Object Definition Table (ODT) number for the internal-name. (See “Using the PROCESS Statement to Specify Compiler Options” on page 32 for information on how to obtain these listings.) Figure 19 shows the Data Division map, and Figure 20 on page 62 shows the cross-reference listing for the program example, TESTPRT.

Figure 19. Data Division Map for TESTPRT

```
FILE SECTION uses 20 bytes of storage
WORKING-STORAGE SECTION uses 75 bytes of storage
```

The I-NAME for NAME-FIELD
Figure 20. Section of IRP Cross-Reference Listing for TESTPRT

<table>
<thead>
<tr>
<th>ODT</th>
<th>ODT Name</th>
<th>SEQ</th>
<th>Cross Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>0184</td>
<td>.DMPFBH1</td>
<td>514*</td>
<td></td>
</tr>
<tr>
<td>0185</td>
<td>.DMPFBH2</td>
<td>515*</td>
<td></td>
</tr>
<tr>
<td>0104</td>
<td>.DMPFBIB</td>
<td>452*</td>
<td></td>
</tr>
<tr>
<td>0148</td>
<td>.DMPFBLN</td>
<td>445*</td>
<td></td>
</tr>
<tr>
<td>0158</td>
<td>.DMPFBLD</td>
<td>471*</td>
<td></td>
</tr>
<tr>
<td>0186</td>
<td>.DMPFBLP</td>
<td>512 516*</td>
<td></td>
</tr>
<tr>
<td>0182</td>
<td>.DMPFBS</td>
<td>512*</td>
<td></td>
</tr>
<tr>
<td>014C</td>
<td>.DMPFBL1</td>
<td>449*</td>
<td>1065 1066</td>
</tr>
<tr>
<td>014D</td>
<td>.DMPFBL2</td>
<td>450*</td>
<td></td>
</tr>
<tr>
<td>0160</td>
<td>.DMPFBMF</td>
<td>476*</td>
<td></td>
</tr>
<tr>
<td>014E</td>
<td>.DMPFBMN</td>
<td>451*</td>
<td>995 1098 1099 1118 1119</td>
</tr>
<tr>
<td>0180</td>
<td>.DMPFEND</td>
<td>509*</td>
<td></td>
</tr>
<tr>
<td>0150</td>
<td>.DMPFBBD</td>
<td>453*</td>
<td></td>
</tr>
<tr>
<td>015A</td>
<td>.DMPFBBO</td>
<td>470*</td>
<td></td>
</tr>
<tr>
<td>0152</td>
<td>.DMPFBBL</td>
<td>458*</td>
<td></td>
</tr>
<tr>
<td>015F</td>
<td>.DMPFBPO</td>
<td>475*</td>
<td></td>
</tr>
<tr>
<td>0161</td>
<td>.DMPFBQQ</td>
<td>477*</td>
<td></td>
</tr>
<tr>
<td>0155</td>
<td>.DMPFBRC</td>
<td>461*</td>
<td></td>
</tr>
<tr>
<td>0153</td>
<td>.DMPFBRW</td>
<td>459*</td>
<td></td>
</tr>
<tr>
<td>015B</td>
<td>.DMPFBRN</td>
<td>471*</td>
<td></td>
</tr>
<tr>
<td>01F9</td>
<td>.DMPFBSF</td>
<td>446*</td>
<td></td>
</tr>
<tr>
<td>014A</td>
<td>.DMPFBSL</td>
<td>447*</td>
<td></td>
</tr>
<tr>
<td>014B</td>
<td>.DMPFBSN</td>
<td>448*</td>
<td></td>
</tr>
<tr>
<td>014C</td>
<td>.DMPFBTE</td>
<td>449*</td>
<td></td>
</tr>
<tr>
<td>0159</td>
<td>.DMPFBFD</td>
<td>469*</td>
<td></td>
</tr>
<tr>
<td>0183</td>
<td>.DMPFBVL</td>
<td>513*</td>
<td></td>
</tr>
<tr>
<td>0188</td>
<td>.DMPFIOFB</td>
<td>522*</td>
<td></td>
</tr>
<tr>
<td>014D</td>
<td>.DMPFIOFS</td>
<td>545*</td>
<td>546 547</td>
</tr>
<tr>
<td>01A6</td>
<td>.DMPKLYN</td>
<td>551*</td>
<td></td>
</tr>
<tr>
<td>0165</td>
<td>.DMPKDEV</td>
<td>481*</td>
<td>1087 1145</td>
</tr>
<tr>
<td>0144</td>
<td>.DMPFBJS</td>
<td>441*</td>
<td>442 443 444 445 446 447 448 449 450 451 452 453 454 458 459 460 461 462 467 468 469 470 471 472 473 474 475 476 477 479 480 481 482 508 509 510</td>
</tr>
<tr>
<td>01AA</td>
<td>.DMPRCD</td>
<td>555*</td>
<td></td>
</tr>
<tr>
<td>01AC</td>
<td>.DMPRCDN</td>
<td>557*</td>
<td></td>
</tr>
<tr>
<td>01A1</td>
<td>.DMPRCDP</td>
<td>559*</td>
<td></td>
</tr>
<tr>
<td>01A7</td>
<td>.DMPRPM</td>
<td>546*</td>
<td></td>
</tr>
<tr>
<td>015A</td>
<td>.DMPRAON</td>
<td>552*</td>
<td></td>
</tr>
<tr>
<td>0220</td>
<td>.DMPFSC</td>
<td>550*</td>
<td></td>
</tr>
<tr>
<td>0221</td>
<td>.DMPFSCB</td>
<td>686*</td>
<td></td>
</tr>
<tr>
<td>0222</td>
<td>.DMPFSCB</td>
<td>687*</td>
<td>767 914 916</td>
</tr>
<tr>
<td>0223</td>
<td>.DMPFSCC</td>
<td>688*</td>
<td></td>
</tr>
<tr>
<td>0224</td>
<td>.DMPFSCD</td>
<td>670*</td>
<td></td>
</tr>
<tr>
<td>0225</td>
<td>.DMPFSCF</td>
<td>689*</td>
<td>789 904</td>
</tr>
<tr>
<td>0226</td>
<td>.DMPFSCG</td>
<td>671*</td>
<td>672 673 676 677</td>
</tr>
<tr>
<td>0227</td>
<td>.DMPFSCH</td>
<td>672*</td>
<td>753 757 761 765 815</td>
</tr>
<tr>
<td>0228</td>
<td>.DMPFSCI</td>
<td>675*</td>
<td>763</td>
</tr>
<tr>
<td>0229</td>
<td>.DMPFSCJ</td>
<td>676*</td>
<td>674</td>
</tr>
<tr>
<td>0230</td>
<td>.DMPFSCK</td>
<td>677*</td>
<td>678</td>
</tr>
<tr>
<td>021B</td>
<td>.DMPFSC</td>
<td>678*</td>
<td>754 758 769</td>
</tr>
<tr>
<td>022B</td>
<td>.DMPFSCD</td>
<td>680*</td>
<td>681 682 683 684 685 686 687 688 778 789 904</td>
</tr>
<tr>
<td>021A</td>
<td>.DMPFSCF</td>
<td>683*</td>
<td>767</td>
</tr>
<tr>
<td>021C</td>
<td>.DMPFSCG</td>
<td>684*</td>
<td></td>
</tr>
<tr>
<td>021D</td>
<td>.DMPFSCH</td>
<td>685*</td>
<td></td>
</tr>
<tr>
<td>021E</td>
<td>.DMPFSCJ</td>
<td>686*</td>
<td></td>
</tr>
</tbody>
</table>

021C is the ODT number for NAME-FIELD

Now you can use ODT number 021C (for NAME-FIELD), with the following commands, to add a breakpoint to the program example at statement 52.

```
STRDBG TESTPRT
ADDBKP STMT(52)
PGMVAR(’/021C’)
```

These commands are explained in the CL Reference.
The following is displayed when this breakpoint is reached:

```
Display Breakpoint
Statement/Instruction . . . . . . . . . : 52 /0056
Program . . . . . . . . . . . . . . . . : TESTPRT
Recursion level . . . . . . . . . . . : 1
Start position . . . . . . . . . . . : 1
Format . . . . . . . . . . . . . . . . : *CHAR
Length . . . . . . . . . . . . . . . . : *DCL
proc=display.
Variable . . . . . . . . . . . . . . . : /021C
Type . . . . . . . . . . . . . . . . . : CHARACTER
Length . . . . . . . . . . . . . . . . : 2
*:...+....1....+....2....+....3....+....4....+....5
'Z'
```

Press Enter to continue.

F3=Exit Program  F10=Command entry

**Figure 21. Breakpoint at Statement 52**

**Changing Program Variables**

Now you can change the value of program variables to alter your program's processing. You can use the Change Program Variable (CHGPGMVAR) command to change the value of a variable. This procedure is explained in more detail in the CL Reference.

You can use the DSPPGMVAR command to display pointer data items, but you cannot use CHGPGMVAR to change pointer data items. To change pointer data items, you use the CHGHLLPTR or CHG PTR commands. For more information on the CHGHLLPTR and CHG PTR commands, refer to the CL Reference.

**Considerations for Using Breakpoints**

You should know the following breakpoint characteristics before using breakpoints:

- If a breakpoint is bypassed by, for example the GO TO statement, that breakpoint isn't processed.
- When a breakpoint is set on a statement, the breakpoint occurs before that statement is processed.
- Breakpoint functions are specified through OS/400 commands.

These functions include:

- Adding breakpoints to programs
- Removing breakpoints from programs
- Displaying breakpoint information
- Resuming the running of a program after a breakpoint has been reached (displayed).
See the *CL Programmer’s Guide* for descriptions of these commands and for more details about breakpoints.

## Using a Trace

A trace is a record of some or all of the statements run in a program. If requested, a trace also records the values of specific variables used in the program statements.

<table>
<thead>
<tr>
<th>Program</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td>Processing Order</td>
</tr>
<tr>
<td>1 ......</td>
<td>1</td>
</tr>
<tr>
<td>2 ......</td>
<td>6</td>
</tr>
<tr>
<td>3 ......</td>
<td>7</td>
</tr>
<tr>
<td>4 ......</td>
<td>8</td>
</tr>
<tr>
<td>5 ......</td>
<td>6</td>
</tr>
<tr>
<td>6 ......</td>
<td>7</td>
</tr>
<tr>
<td>7 ......</td>
<td>2</td>
</tr>
<tr>
<td>8 ......</td>
<td>6</td>
</tr>
<tr>
<td>.</td>
<td>7</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

A trace differs from a breakpoint because the number of statements involved in the trace affects where the trace will end. The system records all the traced statements that were processed. You can request a display of the traced information, which shows the sequence in which the statements were processed and, if requested, the values of the variables used in the statements.

You specify which statements the system will trace. You can also specify that variables be displayed only when their value has changed since the last trace statement was run.

You can specify a trace of one statement in a program, a group of statements in a program, or all the statements in an entire program.

### Example of Using a Trace

Figure 22 on page 65 shows a portion of a COBOL program example, TESTPRT. The following OS/400 command adds a trace of statements 54 through 58 in that program. The variable NO-OF-DEPENDENTS is to be recorded only if its value changes between statements 54 and 58:

```
ADDTRC STMT((54 58))
PGMVAR('NO-OF-DEPENDENTS')
OUTVAR(*CHG)
```

**Note:** STRDBG must be entered before the ADDTRC statement.
Figure 22. Example of a COBOL Program Using a Trace

Figure 23 on page 66 is an example of a listing of the traced information. This information is produced by the Display Trace Data (DSPTRCDTA) command:

DSPTRCDTA OUTPUT(*PRINT) CLEAR(*YES)

This command is explained in the CL Reference.
## Considerations for Using a Trace

You should understand the following trace characteristics before using them:

- Statements bypassed by, for example the GO TO statement, are not included in the trace.
- Trace functions are specified through OS/400 commands in the job containing the traced program. These functions include adding trace requests to a
program, removing trace requests from a program, removing data collected from previous traces, displaying trace information, and displaying the traces that have been specified for a program.

- In addition to statement numbers, names of COBOL-generated routines can appear on the trace output STMT field.

See the CL Programmer’s Guide for more information on traces.

Using a Debug Run-Time Switch

A run-time switch is provided for the COBOL Debug facility. This switch activates the debugging code generated when WITH DEBUGGING MODE is specified. When the switch is set on, all compiled debugging sections are activated; when it is set off (the default), the USE FOR DEBUGGING Declarative procedures are deactivated. Refer to Appendix B, “Debugging Features” on page 313 for more information on COBOL debugging features and the use of the run-time switch.

Using a COBOL Formatted Dump

Some COBOL run-time messages allow you to obtain a COBOL formatted dump option by selecting either D or F. The formatted dump (choose D) includes current information about the files in your program, contents of fields, data structures, arrays, and tables for user-defined COBOL data variables.

If you choose the F option, the dump also includes a list of compiler-generated fields and their contents.

Both the D option and the F option will dump the first 256 characters of program variables. Any variable greater than 256 characters will be truncated.

If you do not want a dump, specify C (cancel with no dump). Reply C is also the default reply for all COBOL inquiry messages that allow a dump.

For more information about reply modes see “Replying to Run-Time Inquiry Messages” on page 52.

The output for the dump is sent to the IBM-supplied printer file QPPGMMDMP.

To see an example of a formatted dump, refer to Appendix H, “Example of a COBOL Formatted Dump” on page 371.
Chapter 6. COBOL/400 Exception and Error Handling

This chapter describes COBOL/400 error handling and its use. It also explains the relationship between error handling and the processing of I/O verbs.

The COBOL/400 compiler provides two error-handling methods: standard and non-standard. Standard error handling is not available on compilers released earlier than Version 1 Release 3.

Standard Error Handling

Standard error handling gives you extra compatibility with other IBM COBOL compilers (such as VS COBOL II) as well as non-IBM COBOL compilers. It can help you during the processing of I/O statements by catching severe errors that might not otherwise be noticed.

An important characteristic of standard error handling is the issuing of a run-time message when an error occurs during the processing of an I/O statement if there is no AT END/INVALID KEY phrase in the I/O statement, USE procedure for the file, or FILE STATUS clause in the SELECT statement for the file.

Release Sensitivity!

Standard error handling was introduced in Version 1 Release 3 as a default option. To get the error handling that was used in earlier releases, specify *NOSTDERR as a generation option of the CRTCBPGM command, or NOSTDERR in the PROCESS statement.

Error Handling Overview

When you run a COBOL program, several types of errors can occur. The COBOL statement active at the time of a given error causes certain COBOL clauses or phrases to run.

During arithmetic operations, typical errors are size (MCH1210) errors and decimal data (MCH1202) errors; the corresponding error-handling phrase is the SIZE ERROR phrase.

Most MCH errors are not directly detected by COBOL; they are detected by the operating system and result in system messages. COBOL then monitors for these messages, setting internal bits that determine whether to run a SIZE ERROR imperative statement or issue a run-time message (LBE7200) to end the program.

COBOL does detect errors that result from division by zero during an arithmetic operation. If detected by COBOL, these errors cause the SIZE ERROR imperative statement to run.

System message MCH1210 occurs when you move a numeric field to a receiver that is too small. This error is monitored by COBOL, and also results in the running of the SIZE ERROR imperative statement.
LBE7200 is a run-time message that is usually issued when an unmonitored severe error occurs in your COBOL program. Under *NOSTDERR, it can also be issued when an error occurs in the absence of an appropriate error handler.

System message MCH1202 is a typical example of an unmonitored severe error. This kind of error results in the COBOL run-time message LBE7200 (or LBE7204 if the error occurs in a program called by a COBOL program). System messages MCH3601 and MCH0601 are other examples of unmonitored severe errors.

For I/O operations, there are several important error handling phrases and clauses. These are the AT END/INVALID KEY and NO DATA phrases (coded at the COBOL statement level), the USE procedure, and the FILE STATUS clause (coded at the file level). During arithmetic and I/O operations, errors are detected by the system, which sends messages; the messages are then monitored by COBOL. Similar to the case of an error that results from division by zero, COBOL does detect some errors during an I/O operation. Regardless of how an error is detected during an I/O operation, the result will always be an internal file status in which the first character is not zero, run-time message, or both.

General-Use Programming Interface

Using Error-Handling Application Programming Interfaces (APIs)

You can use COBOL/400 APIs to control error handling for you within your programs. These APIs are Retrieve COBOL Error Handler (QLRRTVCE), and Set COBOL Error Handler (QLRSETCE).

The Retrieve COBOL Error Handler (QLRRTVCE) API allows you to retrieve the name of the current or pending COBOL error-handling program. You can call it from any programming language.

The Set COBOL Error Handler (QLRSETCE) API allows you to specify the identity of a COBOL error-handling program. You can call it from any programming language.

You can also use the Change COBOL Main Program (QLRCHGCM) API to create multiple run units, each with its own error handler.

For detailed information on all of these APIs, refer to the System Programmer’s Interface Reference.

Internal and External File Status

You must provide a FILE-CONTROL entry to specify the organization and access method for each file used by your COBOL program. You can also code a FILE STATUS clause in this entry.

The FILE STATUS clause designates one or two data items (coded in the WORKING-STORAGE section) to hold a copy of the result of an I/O operation. Your copy of the first of these items is called the external file status. If you use a TRANSACTION file, you have a further record of the result called the external return code, which consists of the external major and minor return codes.
COBOL keeps its own copies of these two data items, both of which are stored in
the COBOL File Information Block (FIB). In this chapter, file status and
(major/minor) return code refer to COBOL’s copies unless otherwise specified.

During the processing of an I/O statement, the file status can be updated in one of
three ways, as described below. The contents of the file status determine which
error handling procedures to run.

Error handling procedures take control after an unsuccessful input or output opera-
tion, which is denoted by any file status in which the first character is not zero.
Before any of these procedures run, the file status is copied into the external file
status.

The file status is set in one of three ways:

- Method A (all files):
  COBOL checks the contents of variables in file control blocks. If the contents
  are not what is expected, a file status of other than zero is set. Most file
  statuses set in this way result from checking the COBOL File Information Block
  (FIB) and the system User File Control Block (UFCB).

- Method B (transaction files):
  COBOL checks the major and minor return codes from the system. If the major
  return code is not zero, the return code (consisting of major and minor return
  codes) is translated into a file status. If the major return code is zero, the file
  status may have been set by Method A or C.

  Note that for subfile READ, WRITE, and REWRITE operations, only Methods A
  and C apply.

  For a list of return codes and their corresponding file statuses, see “File Struc-
  ture Support Summary and Status Key Values” in the COBOL/400 Reference.

- Method C (all files):
  A message is sent by the system when COBOL calls on data management to
  perform an I/O operation. COBOL then monitors for these messages and sets
  a file status accordingly.

  COBOL specifically monitors for a message by generating message monitors in
  the program object produced at compilation time. Message monitor generation
  is based on the types of files (organization type and access type are examples)
  that you specify in a program. Thus, a message that is specifically monitored
  for in one program may fall under the generic I/O handler in another. More
  information about message monitor generation will follow in this chapter.

  COBOL monitors for most messages sent by the system in response to an I/O
  operation. Typical I/O exceptions result in CPF messages that begin with
  “CPF4” or “CPF5,” and COBOL does specific monitoring for these.

  For a list of messages for which COBOL does specific monitoring, see “File
  Structure Support Summary and Status Key Values” in the COBOL/400
  Reference.
General Error Detection

How File Status is Set

001
Start of I/O operation
– Method A: Check contents of variables in file control blocks.
  (Check, for example, that the file has been opened properly and in the right mode.)

Has internal file status been changed from 00?
Yes  No

002
Call on data management to perform I/O operation
– Method C: Monitor for messages sent by data management and set internal file status accordingly.
– Method A: Check system information blocks and set internal file status if not already set using Method C.

Is the file a transaction file?
Yes  No

003
Set external file status
– Move internal file status to external file status (specified in file status clause). Based on internal file status, run error handling code.

004
Check major and minor return codes from system
– Method B: If data management has sent a message, translate major and minor return codes associated with system message into internal file status.

Continue at Step 003

005
Continue at Step 003
Message Monitor Generation

A message monitor provides a way for a program to handle messages sent by the system or by another program. A message monitor can handle one or more messages.

In some respects, a message monitor resembles a USE procedure. Similar to the way in which a USE procedure specifies actions to take in response to an I/O error, a message monitor specifies an action to take when an error occurs during the processing of a machine interface (MI) instruction. Note that an MI instructional error is signalled by a system message, and note that each COBOL statement is composed of one or more MI instructions.

Unlike a USE procedure (which may not be active during an entire program), a COBOL message monitor becomes active as soon as the program starts. Message monitors set file statuses and indicate SIZE ERROR, END-OF-PAGE, and OVERFLOW conditions.

Message monitors generated by COBOL are grouped into several sets, generated under certain conditions within a COBOL program. The following table provides general guidelines regarding the generation of message monitors:

<table>
<thead>
<tr>
<th>Cause of Message Monitor</th>
<th>Sample Members of Monitored Message Set</th>
</tr>
</thead>
</table>
| You code a file status clause | • File not found, external file status 35  
• Permanent error condition, external file status 30  
• OPEN mode not valid, external file status 37  
• No next record, system message CPF5183 (part of external file status 46)  
• Undefined or unauthorized access type, external file status 91  
• Logic error, external file status 92 (except for system messages CPF4740 and CPF5070)  
• Record is locked, external file status 9D  
• OPEN with commitment control failed, external file status 9P  
• WRITE not valid, system messages CPF5018 and CPF5272 (part of external file status 24). |
| You code an AT END phrase | • End-of-file handler, system messages CPF5001 and CPF5025  
• File not found, external file status 35. |
| You specify a subfile in your program | • Last record written to subfile, external file status 9M or 0M  
• Subfile record not found, system message CPF5020 (part of external file status 23)  
• Subfile boundary violation, system messages CPF5021 and CPF5043 (part of external file status 24). A **boundary violation** is an attempt to write beyond the externally defined boundaries of a sequential file. |
### Table 1 (Page 2 of 2). Generation of Message Monitors

<table>
<thead>
<tr>
<th>Cause of Message Monitor</th>
<th>Sample Members of Monitored Message Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>You code a subfile READ statement with the NEXT MODIFIED phrase</td>
<td>• No modified subfile record, external file status 12.</td>
</tr>
<tr>
<td>You use an indexed sequential file</td>
<td>• No specific monitor (Method A), set internal file statuses 21 and 22.</td>
</tr>
<tr>
<td>There is a keyed READ operation</td>
<td>• System messages CPF5006 and CPF5013 (part of external file status 23).</td>
</tr>
<tr>
<td>There is a sequential WRITE operation</td>
<td>• Boundary violation, system message CPF5116 (part of external file status 34).</td>
</tr>
<tr>
<td>There is an indexed sequential REWRITE operation</td>
<td>• No specific monitor (Method A), set internal file statuses 21, 43, 44, and 9S.</td>
</tr>
<tr>
<td>There is TRANSACTION I/O</td>
<td>• READ timeout, system message CPF4743, set internal file status 00&lt;br&gt;• No data during READ, system message CPF4742, set NO DATA bit&lt;br&gt;• No acquired devices, system message CPF5070 (part of external file status 92)&lt;br&gt;• No devices invited/acquired, system message CPF4740 (part of external file status 92 and external file status 10)&lt;br&gt;• Cancel job, external file status 9A&lt;br&gt;• WRITE failed, external file status 9I&lt;br&gt;• Temporary error, external file status 9N.</td>
</tr>
<tr>
<td>You specify a format clause in an I/O statement</td>
<td>• Format name not valid/not found, internal file status 9K.</td>
</tr>
<tr>
<td>There is any I/O at all (including extended ACCEPT/DISPLAY operations) in your program.</td>
<td>• END-OF-PAGE exception handler (system message CPF5004)&lt;br&gt;• Level check error, external file status 39&lt;br&gt;• Generic exception handler, external file status 90&lt;br&gt;• Indicator mismatch (run-time message LBE7421, system message CPF4238)&lt;br&gt;• Ignore COMMIT or ROLLBACK (system message CPF8350)&lt;br&gt;• Duplicate key, external file status 22.&lt;br&gt;• READ DYNAMIC invalid change of direction, internal file status 9U, system message CPF5184.</td>
</tr>
</tbody>
</table>

**Note:** For a list of monitored messages that fall under a particular external file status, see “File Structure Support Summary and Status Key Values” in the COBOL/400 Reference.

### Ending of a COBOL Program

There are three things that can cause a COBOL program to end:

- A COBOL statement (EXIT PROGRAM, STOP RUN, or GOBACK)
- A reply to an inquiry message
- An implicit STOP RUN or EXIT PROGRAM statement.
A STOP RUN statement is implied when a main COBOL program has no next executable statement (implicit EXIT PROGRAM for a COBOL subprogram), that is, when processing falls through the last statement of a program.

Inquiry messages can be issued in response to a COBOL statement (namely a STOP literal), but they are usually issued when a severe error occurs in a program, or when a COBOL operation does not complete successfully. (Examples are LBE7205, LBE7207, and LBE7208.)

There are four common replies to a COBOL inquiry message: C, D, F, and G (cancel, cancel and dump, cancel and full dump, continue). The first three cause (as their final steps) an implicit STOP RUN followed by escape message LBE9001. LBE9001 indicates that the program is ending because of a message.

An implicit or explicit STOP RUN statement, or a GOBACK statement that appears in a main program, ends the entire COBOL run unit. If an escape message (LBE9001) is issued as the final step of a run unit, the caller of the first COBOL program can monitor for it. (This is because the first COBOL program to be called becomes the main program.)

If a COBOL run unit consists of several COBOL and non-COBOL programs, it is the main COBOL program that can issue the escape message. Thus, any non-COBOL program that is called after the main program cannot monitor for the escape message.

**Return Codes**

When you specify a TRANSACTION file in your program, the FILE STATUS clause of your SELECT statement can contain two data names: the external file status, and the (major and minor) return code. As described under "Internal and External File Status" on page 70, a file status can be set in one of three ways; however, return codes are set by the system after any transaction I/O that calls data management. Consequently, most error conditions that result in a system message also have an associated return code.

Return codes are similar to file status values. That is, CPF messages sent by the system are grouped together under message monitors, and each message monitor sets one or more file statuses.

Similarly, CPF messages are grouped together, and each group of messages generates the same major return code. (The minor return code is not necessarily the same.)

The main difference between file statuses and return codes is that the grouping of CPF messages is different.

Although COBOL only sets return codes for TRANSACTION files, other types of files (such as printer files) also set return codes. You can access the return codes for these files through an ACCEPT from I-O-FEEDBACK operation.
Standard and Nonstandard Error Handling Models

Figures 24 and 25 show the two different error handling models.

Figure 24 (Part 1 of 2). Standard (default) Error Handling

Note:  = Go to  on next page
Figure 24 (Part 2 of 2). Standard (default) Error Handling
Figure 25. Nonstandard Error Handling (available through *NOSTDERR option)
Other I/O exceptions may occur that COBOL does not expect. These also result in CPF4xxx and CPF5xxx messages, but there is not specific monitoring for them. Instead, they are caught by a generic I/O error handler. This error handler monitors for certain ranges of CPF4xxx and CPF5xxx messages; it sets the file status to 90 and follows the Yes branch from position *3 in Figure 25 on page 78.

An I/O exception may occur that is being specifically monitored for and which, according to the nonstandard error handling model, is severe enough to stop the program. In this situation no file status is set.

These I/O exceptions result in specific COBOL escape messages followed by an ending of the program; they follow the Yes branch from position *2 in Figure 25.

Example: CPF4238 - INDARA mismatch between program and file

There is specific monitoring for this message, and the result is error message LBE7021 followed by an ending of the program.

Other COBOL messages that fall into this category are LBE7020 and LBE7022.

During an I/O operation, a problem may occur that is not expected by the system. These problems generally result in messages (such as those starting with “MCH”) that fall outside the CPF4xxx and CPF5xxx range. Such errors, known as unmonitored severe errors, follow the Yes branch from position *1 in Figure 25. These errors are handled by an all-purpose message monitor and result in an ending of the COBOL program. No file status is set.

**Effects of *STDERR and *NOSTDERR on File Status**

- Effects of LBE742x and LBE702x messages:
  
  With *STDERR, file status 90 is set following the issue of LBE742x messages. The program then continues if there is a USE procedure or a FILE STATUS clause.

  With *NOSTDERR, LBE702x messages cause the program to end without setting a file status.

- Ending of a program because of file status 9P or 90:
  
  With *STDERR, a file status of 9P or 90 arising from an I/O error (signalled by CPF4xxx and CPF5xxx messages) does not cause the program to end as long as there is a USE procedure or a FILE STATUS clause. If neither exists, error message LBE7207 is issued.

  With *NOSTDERR, a file status of 9P or 90 in the absence of a USE procedure causes error message LBE7200 to be issued.

- Issuing of an error message for any file status in which the first character is not zero when there is no error handler or FILE STATUS clause:
  
  With *STDERR, any file status in which the first character is not zero when there is no AT END/INVALID KEY phrase, USE procedure, or FILE STATUS clause causes inquiry message LBE7207 (with response options C, D, F, and G) to be issued.

  With *NOSTDERR, any file status in which the first character is not zero when there is no AT END/INVALID KEY phrase or USE procedure allows the program to continue unless it has already ended.
Processing of I/O Verbs

The following diagram shows when the USE procedure and the (NOT) AT END, (NOT) INVALID KEY, and NO DATA imperative statements are run. This has been in place since Version 1 Release 3, and is independent of the error handling method you choose (*STDERR or *NOSTDERR).

Note that the file status shown here refers to the internal file status.

---

**Figure 26 (Part 1 of 2). Processing of I/O Verbs**

---

Is there an AT END phrase?

Is the leftmost character of file status equal to 2?

Is there a USE procedure?

Run AT END imperative statement

Run USE procedure

Continue COBOL program

---

Note: E1 - Go to E1 on next page
Figure 26 (Part 2 of 2). Processing of I/O Verbs

Note: Follow the parts of the diagram that apply to your statements.

Common Exceptions and Some of Their Causes

MCH1202 Decimal data error:

- A numeric elementary item has been used as a source when no valid data has been previously stored in it. The item should have a VALUE clause, or a MOVE statement should be used to initialize its value.
- An attempt has been made to place nonnumeric data in a numeric item.
- Bad data was written to a subfile earlier in the program. The subfile data is not validated until it is written to the display, so the 1202 error can occur on the WRITE of a subfile control record, but the bad data was actually put to the subfile earlier.

MCH0601 Pointer exceptions:

- Part of a linkage section item extended beyond the space allocated.

For example, if you set the address of a linkage section item, and one or more of its elementary data items extend beyond the space with a MOVE to the elementary data item, MCH0601 is issued.
For more information on using pointers, refer to “Using Pointers in a COBOL/400 Program” on page 282.

MCH0602 Pointer alignment:
- The pointer alignment in the Working-Storage Section of the calling program does not match the alignment in the Linkage Section of the called program. Alignment must be on a 16-byte boundary.

For more information on using pointers, refer to “Using Pointers in a COBOL/400 Program” on page 282.

MCH3601 Pointer error:
- A reference is made to a record or a field within a record and the associated file has been closed or has never been opened.

For example, the OPEN for the file was unsuccessful and the processing of any other I/O statement for that file is attempted. The file status should be checked before any other I/O is attempted.

CPF2415 End of requests:
- An attempt has been made to accept input from the job input stream while the system is running in batch mode and no input is available.

Recovery After a Failure

Recovery with Commitment Control

When the system is restarted after a failure, files under commitment control are automatically restored to their status at the last commitment boundary. For additional information about commitment control, see “Commitment Control Considerations” on page 94.

For a job failure (either because of user or system error), files under commitment control are restored as part of job termination to the files’ status at the previous commitment boundary.

Because files under commitment control are rolled back after system or process failure, this feature can be used to help in restarting. You can create a separate record to store data that may be useful should it become necessary to restart a job. This restart data can include items such as totals, counters, record key values, relative key values, and other relevant processing information from an application.

If you keep the restart data mentioned above in a file under commitment control, the restart data will also be permanently stored in the database when a COMMIT statement is issued. When a ROLLBACK occurs after job or process failure, you can retrieve a record of the extent of processing successfully processed before failure. Note that the above method is only a suggested programming technique and will not always be suitable, depending on the application.
TRANSACTION File Recovery

In some cases, you can recover from I/O errors on TRANSACTION files without intervention by the operator, or the varying off/varying on of work stations or communications devices.

For potentially recoverable I/O errors on TRANSACTION files, the system initiates action in addition to the steps that must be taken in the application program to attempt error recovery. For more information about action taken by the system, see the Remote Work Station Guide.

By examining the file status after an I/O operation, the application program can determine whether a recovery from an I/O error on the TRANSACTION file is possible. If the File Status Key has a value of 9N, the application program may be able to recover from the I/O error. A recovery procedure must be coded as part of the application program and varies depending on whether a single device was acquired by the TRANSACTION file or whether multiple devices were attached.

For a file with one acquired device:

1. Close the TRANSACTION file with the I/O error.
2. Reopen the file.
3. Process the steps necessary to retry the failing I/O operation. This may involve a number of steps, depending on the type of program device used. (For example, if the last I/O operation was a READ, you may have to repeat one or more WRITE statements, which were processed prior to the READ statement.)
   For more information on recovery procedures, see the ICF Programmer’s Guide.

For a display file with multiple devices acquired:

1. DROP the program device that caused the I/O error on the TRANSACTION file.
2. ACQUIRE the same program device.
3. See Step 3 above.

For an ICF file with multiple devices acquired:

1. ACQUIRE the same program device.
2. See Step 3 above.

For a display file with multiple devices acquired:

Application program recovery attempts should typically be tried only once.

If the recovery attempt fails:

- If the file has only one program device attached, terminate the program through processing of the STOP RUN, EXIT PROGRAM, or GOBACK statement, and attempt to locate the source of the error.
- If the file has multiple acquired program devices, you may want to do one of the following:
  - Continue processing without the program device that caused the I/O error on the TRANSACTION file, and reacquire the device later.
  - End the program.
For a description of major and minor return codes that may help in diagnosing I/O errors on the TRANSACTION file, see the ICF Programmer’s Guide or the Data Management Guide.

Figure 27 gives an example of an error recovery procedure.

---

**Figure 27. Example of Error Recovery Procedure -- DDS**

---

For a description of major and minor return codes that may help in diagnosing I/O errors on the TRANSACTION file, see the ICF Programmer’s Guide or the Data Management Guide.

---

**Figure 27. Example of Error Recovery Procedure -- DDS**

---
Figure 28 (Part 1 of 3). Example of Error Recovery Procedure
Figure 28 (Part 2 of 3). Example of Error Recovery Procedure
This defines processing that takes place when an I/O error occurs on RECOVFILE.

This prints out information to help in diagnosing the problem.

If the file-status equals 9N (temporary error), and no previous error recovery has been attempted for this I/O operation, error recovery is now attempted.

To avoid program looping, recovery is not attempted now if it was attempted previously.

Recovery consists of dropping, then reacquiring, the program device on which the I/O error occurred.

The mainline of the program consists of writing to and reading from a device until the user signals an end to the program by pressing F1.

If the WRITE operation failed but recovery was done, the WRITE is attempted again.

If the READ operation failed, processing will continue by writing to the device again, and then attempting the READ again.
Chapter 7. File and Data Management

This chapter contains general file and data management information you may need when creating COBOL/400 applications.

This chapter describes:
- The device-independent and device-dependent characteristics of COBOL/400 programs on the AS/400 system
- Input and output spooling functions
- System override considerations
- File and record locking considerations
- Commitment control
- Unblocking and blocking records
- File status and feedback areas
- General information about the use of program-described files and externally described files in a COBOL/400 program
- The Format 2 COPY statement (DD, DDR, DDS, or DDSR option).

The maximum number of files that you can define and open within number of files used by a program a COBOL program is 99. If you use extended display options, the maximum number is 98. For information on specifying the extended display options, refer to page 23.

Device Independence and Device Dependence

The key element for all I/O operations on the AS/400 system is the file. All files used are defined to the operating system. The operating system maintains a description of each file that is used by a program.

The files are kept online and serve as the connecting link between a program and the device used for I/O. The actual device association is made when the file is processed. In some instances, this type of I/O control allows the user to change the attribute of the file (and, in some cases, change the device) used in a program without changing the program.

In the COBOL/400 language, the file name specified in the ASSIGNMENT-NAME entry of the ASSIGN clause of the file control entry is used to point to the file. This file name points to the system file description:

The COBOL device name in the ASSIGN clause defines the COBOL functions that can be processed on the selected file. At compilation time, certain COBOL func-
tions are valid only for a specific COBOL device name; in this respect, COBOL is
device dependent. The following are examples of device dependency:

- SUBFILE operations are valid only for a WORKSTATION device.
- Indicators are valid only for WORKSTATION or FORMATFILE devices.
- LINAGE is valid only for the PRINTER device.
- OPEN INPUT WITH NO REWIND is valid only for a TAPEFILE device.

For example, assume that the file name FILEY is associated in the COBOL
program with the FORMATFILE device. The device FORMATFILE is an inde-
pendent device type. Therefore, no line or page control specifications are valid in
the COBOL program in the WRITE ADVANCING statement. When the program is
run, the actual I/O device is specified in the description of FILEY. For example, the
device might be a printer; only the default line and page control or those defined in
the DDS would be used:

```
FILEY
DEV(QPRINT)
```

CL commands can be used to override a parameter in the specified file description
or to redirect a file at compilation time or run time. File redirection allows the user
to specify one file at compilation time and another file at run time:

```
FILEX
DEV(QPRINT)
```

Override Command:
```
OVRDKTF FILE(FILEX) TOFILE (FILEA)
```

In the preceding example, the Override to Diskette File command (OVRDKTF)
allows the program to run with an entirely different device file than was specified at
compilation time.

Not all file redirections or overrides are valid. At run time, checking occurs to
ensure that the specifications within the COBOL program are valid for the file being
processed. The OS/400 operating system allows some file redirections even if
device specifics are contained in the program. For example, if the COBOL device
name is PRINTER and the actual file the program uses is not a printer, the oper-
ating system ignores the COBOL print spacing and skipping specifications.
There are other file redirections that the operating system does not allow and that cause program termination. For example, if the COBOL device name is DATABASE or DISK and a keyed READ operation is specified in the program, the program is terminated if the actual file the program uses is not a disk or database file.

See “System Override Considerations” on page 92 for more detailed information on valid file redirections and file overrides.

**Spooling**

The AS/400 system provides for the use of input and output spooling functions. Each AS/400 file description contains a spool attribute that determines whether spooling is used for the file at run time. The COBOL program is not aware that spooling is being used. The actual physical device from which a file is read or to which a file is written is determined by the spool reader or the spool writer. See the Data Management Guide for more detailed information on spooling.

**Output Spool**

Output spooling is valid for batch and interactive jobs. The description of the file that is specified in COBOL by the system-name contains the specification for spooling as shown in the following example:

File override commands can be used at run time to override the spooling options that are specified in the file description, such as the number of copies to be printed. In addition, AS/400 spooling support allows you to redirect a file after the program has run. For example, you can direct the printer output to a different device, such as a diskette.
Input Spool

Input spooling is valid only for inline data files in batch jobs. If the input data read by COBOL comes from a spooled file, COBOL is not aware of which device the data was spooled in from.

The data is read from a spooled inline file:

![Diagram of input spool process]

See the Data Management Guide for more information on inline data files.

System Override Considerations

You must specify any overrides before the file is opened by the COBOL program. The system uses the file override command to determine the file to open and the attributes of the file.

The simplest form of overriding a file is to override some attributes of the file. For example, FILE(OUTPUT) with COPIES(2) is specified when a printer file is created. Then, before the COBOL program is run, the number of printed copies of output can be changed to 3. The override command is as follows:

```
OVRPRTF FILE(OUTPUT) COPIES(3)
```

Another form of file overriding is to redirect the COBOL program to access a different file. When the override redirects the program to a file of the same type (such as a printer file to another printer file), the file is processed in the same manner as the original file.

When the override redirects the program to a file of a different type, the overriding file is processed in the same manner as the original file would have been processed. Device-dependent specifications in the COBOL program are ignored, and the defaults are taken by the system.

*Not all file redirections are valid.* For example, an indexed file for a COBOL program can only be overridden to another indexed file with a keyed access path.

Multiple member processing can be accomplished for a database file by overriding a database file to process all members. Note the following exceptions:

- A database source file used for a COBOL program cannot be overridden to process all members. Specifying OVRDBF MBR(*ALL) will result in the termination of the compilation.
• A database file used for a COPY statement cannot be overridden to process all members. Specifying OVRDBF MBR(*ALL) will cause the COPY statement to be ignored.

The COBOL programmer must ensure that file overrides are applied properly. For more information on valid file redirections, the device dependent characteristics ignored, and the defaults assumed, see the *Data Management Guide*.

---

**File and Record Locking by COBOL**

The operating system allows a lock state (exclusive, exclusive allow read, shared-for-update, shared-no-update, or shared-for-read) to be placed on a file used during a job step. The file can be placed in a lock state with the Allocate Object (ALCOBJ) command.

By default, the operating system places the following lock states on database files when the files are opened by COBOL programs:

<table>
<thead>
<tr>
<th>OPEN Type</th>
<th>Lock State</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
<td>Shared-for-read</td>
</tr>
<tr>
<td>I/O</td>
<td>Shared-for-update</td>
</tr>
<tr>
<td>EXTEND</td>
<td>Shared-for-update</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>Shared-for-update</td>
</tr>
</tbody>
</table>

**EXTEND mode** is a method of adding records to the end of a sequential file when the file is opened.

The shared-for-read lock state allows another user to open the file with a lock state of shared-for-read, shared-for-update, shared-no-update, or exclusive-allow-read, but the user cannot specify the exclusive use of the file. The shared-for-update lock state allows another user to open the file with a shared-for-read or shared-for-update lock state.

The operating system places the shared-for-read lock on the device file and an exclusive-allow-read lock state on the device. Another user can open the file but cannot use the same device.

**Note:** When a COBOL program opens a physical file for OUTPUT, that file will be subject to an exclusive lock for the period of time necessary to clear the member.

For more information on allocating resources and the lock states, see the *Data Management Guide*.

---

**Locking and Releasing Records**

When a database record is read by COBOL and the file is opened for I/O, a lock is placed on that record so that another program cannot update it. That is, the record can be read by another program if it opens a file for input, but not if it opens the file for I/O.

For information about the duration of record lock with and without commitment control, refer to Table 2 on page 96.
To prevent the READ statement from locking records on files opened in I/O (update) mode, you can use the NO LOCK phrase. The READ WITH NO LOCK statement unlocks records locked by a previous READ statement. For more information about this phrase, refer to the section on the READ statement in the COBOL/400 Reference.

For a logical file based on one physical file, the lock is placed on the record in the physical file. If a logical file is based on more than one physical file, a lock is placed on one record in each physical file.

This lock applies not only to other programs, but also to the original program if it attempts to update the same underlying physical record through a second file.

**Note:** When a file with indexed or relative organization is opened for I/O, using random or dynamic access, a failed I/O operation on any of the I/O verbs except WRITE also unlocks the record. A WRITE operation is not considered an update operation; therefore, the record lock is not released.

For more information about releasing database records read for update, see the Data Management Guide.

**Sharing an Open Data Path**

If you have already opened a file through another program in your routing step, your COBOL program can use the same Open Data Path (ODP) to access the file.

**Note:** Routing steps are described in the Programming: Work Management Guide; a job usually contains only one routing step.

The following rules apply to shared ODPs:

1. You must specify SHARE(*YES) in the command that creates the file, in a change command, or in an override command for the file.

2. Once a file with a shared ODP has been opened for the first time by a program and remains open, subsequent OPEN operations within the same routing step run faster than standard OPEN operations. The speed of I/O operations other than opens is not affected.

3. Your use of the file within your different programs should be consistent. For example, if a non-COBOL program performs a READ PREVIOUS operation using blocked I/O, the COBOL READ statement might retrieve the record preceding the current file position rather than the record following the current file position.

**Commitment Control Considerations**

Commitment control is a function that allows:

- Synchronization of changes to database files within the same job
- Cancelation of changes that should not be permanently entered into the database
- Locking of records being changed until changes are complete
- Techniques for recovering from job or system failure.

In some applications, it is desirable to synchronize changes to database records. If the program determines the changes are valid, the changes are then permanently
made to the database (a COMMIT statement is processed). If the changes are not valid, or if a problem occurs during processing, the changes can be canceled (a ROLLBACK statement is processed). (When a file is cleared after being opened for OUTPUT, processing of a ROLLBACK does not restore cleared records to the file.) Changes made to records in a file that is not under commitment control are always permanent. Such changes are never affected by subsequent COMMIT or ROLLBACK statements.

Each point where a COMMIT or ROLLBACK is successfully processed is a commitment boundary. (If no COMMIT or ROLLBACK has yet been issued in a program, a commitment boundary is created by the first open of any file under commitment control.) The committing or rolling back of changes only affects changes made since the previous commitment boundary.

The synchronizing of changes at commitment boundaries makes restart or recovery procedures after a failure easier. For more information, see “Recovery After a Failure” on page 82.

When commitment control is used for database files, records in those files are subject to either a high lock level LCKLVL(*ALL) or a low lock level LCKLVL(*CHG). With a low lock level (*CHG), all records that are changed (rewritten, deleted, or added) in files under commitment control are locked until a COMMIT or ROLLBACK statement is successfully processed. With a high lock level (*ALL), all records accessed, whether for input or output, are locked until a COMMIT or ROLLBACK is successfully processed. For both record locking levels, no other job can modify data in locked records until the COMMIT or ROLLBACK has been successfully completed. (A locked record can only be modified within the same job and through the same physical or logical file.)

The lock level also governs whether locked records can be read. With a high lock level (*ALL), you cannot read locked records in a database file. With a low lock level (*CHG), you can read locked records in a database file, provided the file is opened as INPUT in your job, or opened as I/O and READ WITH NO LOCK is used.

A third lock level can be obtained by specifying LCKLVL(*CS), in which every record accessed from files under commitment control is locked. Records that are not updated or deleted are locked only until a different record is accessed. Records that are updated, added, or deleted are locked until the transaction is committed or rolled back.

Other jobs, where files are not under commitment control, can always read locked records, regardless of the lock level used, provided the files are opened as INPUT. Because it is possible in some cases for other jobs to read locked records, data can be accessed before it is permanently committed to a database. If a ROLLBACK statement is processed after another job has read locked records, the data accessed will not reflect the contents of the database.

Table 2 shows record locking considerations for files with and without commitment control.
A file under commitment control can be closed or opened without affecting the status of changes made since the last commitment boundary. A COMMIT must still be issued to make the changes permanent, or a ROLLBACK issued to cancel the changes. A COMMIT statement, when processed, leaves files in the same open or closed state as before processing.

All files under commitment control within the same job must be journaled to the same journal. For more information about journal management and its related functions, and for more information about commitment control, refer to the Advanced Backup and Recovery Guide.

Commitment control must also be specified outside the COBOL language through the OS/400 control language (CL). The Start Commitment Control (STRCMTCCTL) command establishes the capability for commitment control and sets the level of record locking at the high level (*ALL), or the low level (*CHG). The STRCMTCCTL command does not automatically initiate commitment control for a file. That file must also be specified in the COMMITMENT CONTROL clause of the I-O-CONTROL paragraph within the COBOL program. The commitment control environment is normally ended by using the End Commitment Control

---

### Table 2. Record Locking Considerations with and without Commitment Control

<table>
<thead>
<tr>
<th>VERB</th>
<th>OPEN MODE</th>
<th>LOCK LEVEL</th>
<th>DURATION OF RECORD LOCK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELETE</td>
<td>I-O</td>
<td>Without Commitment Control</td>
<td>DELETE Operation or COMMIT or ROLLBACK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>With Commitment Control</td>
<td>*CHG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>*ALL</td>
</tr>
<tr>
<td>READ</td>
<td>INPUT</td>
<td>Without Commitment Control</td>
<td>READ Operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>With Commitment Control</td>
<td>*CHG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>*ALL</td>
</tr>
<tr>
<td>READ WITH NO LOCK</td>
<td>I-O</td>
<td>Without Commitment Control</td>
<td>READ Operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>With Commitment Control</td>
<td>*CHG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>*ALL</td>
</tr>
<tr>
<td>READ</td>
<td>I-O</td>
<td>Without Commitment Control</td>
<td>READ Operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>With Commitment Control</td>
<td>*CHG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>*ALL</td>
</tr>
<tr>
<td>REWRITE</td>
<td>I-O</td>
<td>Without Commitment Control</td>
<td>REWRITE Operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>With Commitment Control</td>
<td>*CHG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>*ALL</td>
</tr>
<tr>
<td>START</td>
<td>INPUT</td>
<td>Without Commitment Control</td>
<td>START Operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>With Commitment Control</td>
<td>*CHG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>*ALL</td>
</tr>
<tr>
<td>START</td>
<td>I-O</td>
<td>Without Commitment Control</td>
<td>START Operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>With Commitment Control</td>
<td>*CHG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>*ALL</td>
</tr>
<tr>
<td>WRITE</td>
<td>I-O</td>
<td>Without Commitment Control</td>
<td>WRITE Operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>With Commitment Control</td>
<td>*CHG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>*ALL</td>
</tr>
<tr>
<td>WRITE</td>
<td>OUTPUT</td>
<td>Without Commitment Control</td>
<td>WRITE Operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>With Commitment Control</td>
<td>*CHG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>*ALL</td>
</tr>
</tbody>
</table>
(ENDCMTCTL) command. This causes any uncommitted changes for database files under commitment control to be canceled. (An implicit ROLLBACK is processed.) Refer to the CL Reference for more information on the STRCMTCTL and ENDCMTCTL commands.

For more information about commitment control, see the Advanced Backup and Recovery Guide.

Note: The ability to prevent reading of uncommitted data that has been changed is a function of commitment control and is only available if you are running under commitment control. Normal (noncommitted) database support is not changed by the commitment control extension, and allows reading of locked records when a file that is opened only for input is read. Try to use files consistently. Typically, files should always be run under commitment control or never be run under commitment control.

Figure 29 on page 98 illustrates a possible usage of commitment control in a banking environment. The program processes transactions for transferring funds from one account to another. If no problems occur during the transaction, the changes are committed to the database file. If the transfer cannot take place because of improper account number or insufficient funds, a ROLLBACK is issued to cancel the changes.
Figure 29. Example of Use of Commitment Control --DDS
Figure 30 (Part 1 of 3). Example of Use of Commitment Control
Figure 30 (Part 2 of 3). Example of Use of Commitment Control
A separate indicator area is provided for the program.

The COMMITMENT CONTROL clause specifies files to be placed under commitment control. Any files named in this clause are affected by the COMMIT and ROLLBACK verbs.

The Format 2 COPY statement with the indicator attribute INDIC, defines data description entries in WORKING-STORAGE for the indicators to be used in the program.

IN96 is set if there is an invalid file status.

IN95 is set if there is an INVALID KEY condition on the REWRITE operation.

IN99 is set if the entered account number is invalid for the account to which money is being transferred.

IN97 is set if the entered account number is invalid for the account to which money is being transferred.

If an INVALID KEY condition occurs on the READ, a ROLLBACK is used and the record lock placed on the record after the first READ is released.

If the transfer of funds is not allowed (an indicator has been set), the ROLLBACK statement is processed. All changes made to database files under commitment control are canceled.

If the transfer of funds was valid (no indicators have been set), the COMMIT statement is processed, and all changes made to database files under commitment control become permanent.
The INDICATORS phrase is required for options on the work station display that are controlled by indicators.

Unblocking Input Records and Blocking Output Records

A block contains more than one record. In the interest of improving the performance of input and output operations, the COBOL compiler generates code to unblock input records and block output records in either of the following conditions:

1. *NOBLK is specified (with or without a BLOCK CONTAINS clause) and all of the following conditions are met:
   a. ACCESS IS SEQUENTIAL is specified for the file.
   b. The file is opened only for INPUT or OUTPUT in that program.
   c. The file is assigned to DISK, DATABASE, DISKETTE, or TAPEFILE.
   d. No START statements are specified for the file.

   For RELATIVE organization, blocking is not performed for OPEN OUTPUT.

   If you specify BLOCK CONTAINS, it is ignored except for tape files. For tape files, the BLOCK CONTAINS clause controls the number of records to be blocked. If you do not specify BLOCK CONTAINS, the system determines the number of records to be blocked. In the case of DISKETTE files, the system always determines the number of records to be blocked.

2. *BLK is specified with BLOCK CONTAINS and all of the following conditions are met:
   a. ACCESS IS SEQUENTIAL or ACCESS IS DYNAMIC is specified for the file.
   b. The file is opened only for INPUT or OUTPUT in that program.
   c. The file is assigned to DISK, DATABASE, DISKETTE, or TAPEFILE.

   For RELATIVE organization, blocking is not performed for OPEN OUTPUT.

   The BLOCK CONTAINS clause controls the number of records to be blocked. In the case of DISKETTE files, the system always determines the number of records to be blocked.

   Even when all of the above conditions are met, certain OS/400 restrictions can cause blocking and unblocking to not be processed. In these cases, performance improvements will not be realized.

   If you are using dynamically accessed and indexed organization files, you can use READ PRIOR and READ NEXT to perform blocking. When using READ PRIOR and READ NEXT to perform blocking, you cannot change direction while there are records remaining in the block. To clear the records from a block, specify a random operation, such as a random READ or a random START, or use a sequential READ FIRST or READ LAST.

   If an illegal change of direction takes place, file status 9U results. No further I/O is possible until the file is closed and reopened.

   You can override blocking at run time by specifying SEQONLY(*NO) for the OVRDBF command.
For disk and database files, when you use BLOCK CONTAINS, and if the blocking factor of zero is specified or calculated, the system determines the blocking factor.

There are certain instances in which the blocking factor you specify may be changed. See the Database Guide for more information about these situations.

Where a block of records is written or read, the I/O feedback area contains the number of records in that block. The I/O-FEEDBACK area is not updated after each read or write for files where multiple records are blocked and unblocked by COBOL. It is updated when the next block is read or written. See “I/O FEEDBACK” in the COBOL/400 Reference for more information.

For database files, you may not see all changes as they occur, if the changes are made in different programs. For a description of the effect of blocking on changes to database files, see the discussion on sequential-only processing in the Database Guide.

### File Status and Feedback Areas

To transfer data (OPEN-FEEDBACK or I-O-FEEDBACK areas) associated with an open file to an identifier use the following format:

```
ACCEPT identifier FROM mnemonic-name FOR file-name
```

See the “ACCEPT Statement” section of the COBOL/400 Reference for more information on specifying this statement. See the “Attribute Data Formats” section of the COBOL/400 Reference for information on the OPEN-FEEDBACK and the I-O-FEEDBACK areas.

Refer to the Data Management Guide for information on OPEN-FEEDBACK and I-O-FEEDBACK and the layout and description of the data areas contained in the feedback areas.

When the FILE STATUS clause is specified, the system moves a value into the status key data item after each input/output request that explicitly or implicitly refers to this file. This 2-character value indicates the run status of the statement. When the compiler generates code to block output records or unblock input records, file status values that are caused by OS/400 exceptions are set only when a block is processed. For more information about blocking records, refer to “Unblocking Input Records and Blocking Output Records” on page 102.

The I-O-FEEDBACK area is not updated after each read or write for files in which multiple records are blocked and unblocked by COBOL.
For database files, you may not see all changes as they occur, if the changes are made in different programs. For a description of the effect of blocking on changes to database files, see the discussion on Sequential-Only Processing in the Database Guide.

File Descriptions

All files on the AS/400 system are defined to the OS/400 operating system. The extent to which files can be defined differs:

- A program-described file is described at the field level within the COBOL program in the Data Division. The description of the file to the operating system includes information about the type of file and the length of the records in the file.

- An externally described file is described at the field level to the operating system through IDDU, SQL/400* commands, or DDS. If you create a file (for instance, by using the CRTPF command) without specifying DDS for it, the file still has a field description. The single field has the same name as the file, and has the record length you specified in the create command. The description includes information about the type of file, such as database or a device, and a description of each field and its attributes. The file must be created before you compile the program.

Both externally described files and program-described files must be defined in the COBOL program within the INPUT-OUTPUT SECTION and the FILE SECTION. Record descriptions in the FILE SECTION for externally described files can be defined with the Format 2 COPY statement.

Device-dependent functions such as forms control are not extracted by the Format 2 COPY operation. Only field-level descriptions are extracted.

When EXTERNALLY-DESCRIBED-KEY is specified as RECORD KEY, the fields that make up RECORD KEY are also extracted from DDS.

For more information on the Format 2 COPY statement, see Figure 37 on page 112 and the accompanying text.

Note: Actual file processing within the Procedure Division is the same, if the file is externally described or program-described.

Program-Described Files

Records and fields for a program-described file are described by coding record descriptions in the File Section of the COBOL program instead of using the Format 2 COPY statement.

The file must exist on the system before the program can run, except when you use dynamic file creation, by specifying GENOPT(*CRTF) on the CRTCBLPGM command. For more information, refer to the description of the GENOPT parameter on page 22, or the OPEN statement in the COBOL/400 Reference. To create a file, use one of the Create File commands, which can be found in the CL Reference.
DDS can be used with the Create File commands. For a COBOL indexed file, a keyed access path must be created. Specify a key in DDS when the file is created. The record key in COBOL must match the key defined when the file was created.

**Externally Described Files**

Externally described files offer the following advantages over program-described files:

- **Less coding in COBOL programs.** If the same file is used by many programs, the fields can be defined once to the operating system, and then used by all the programs. This eliminates the need to code a separate record description for each program that uses the file.

- **Less maintenance activity when the file’s record format is changed.** You can often update programs by changing the file’s record format and then recompiling the programs that use the file without changing any coding in the program.

- **Improved documentation.** Programs using the same files use consistent record format and field names.

- **Any editing to be processed on externally described output files can be specified in DDS.**

The external description for a file includes:

- The record format specifications that contain a description of the fields in a record

- Access path specifications that describe how the records are to be retrieved.

These specifications come from the external file description and from the OS/400 command you use to create the file.

You can use an externally described file within a program by making it a program-described file (by coding the record description in the source). In this case, the compiler does not copy the external field-level description of the file at compilation time. You may find this useful during conversions, since an existing program can use a program-described file while a new program uses an externally described file to refer to the same file.

Figure 31 on page 106 shows how COBOL programs can relate to files on the AS/400 system, making use of external file descriptions from DDS.
The COBOL program uses the field level description of a file that is defined to the operating system. The COBOL user coded a Format 2 COPY statement for the record description. At compilation time, the compiler copies in the external field-level description and translates it into a syntactically correct COBOL record description. The file must exist at compilation time.

An externally described file is used as a program-described file in the COBOL program. The entire record description for the file is coded in the COBOL program. This file does not have to exist at compilation time.

A file is described to the operating system as far as the record level only. The entire record description must be coded in the COBOL program. This file does not have to exist at compilation time.

A file name can be specified for compilation time, and a different file name can be specified for run time. A COBOL Format 2 COPY statement generates the record description for the file at compilation time. At run time, a different library list or a file override command can be used so that a different file is accessed by the program. The file description copied in at compilation time is used to describe the input records used at run time.

Note: For externally described files, the two file formats must be the same. Otherwise, a level check error will occur.

Data Description Specifications (DDS)

You can use Data Description Specifications (DDS) to describe files at the field level to the operating system. In DDS, each record format in an externally described file is identified by a unique record format name.

The record format specifications describe the fields in a record and the location of the fields in a record. The fields are located in the record in the order specified in DDS. The field description generally includes the field name, the field type (character, binary, external decimal, or internal decimal), and the field length (including the number of decimal positions in a numeric field). Instead of being specified in the record format for a physical or logical file, the field attributes can be defined in a field reference file. (See Figure 32 on page 107.)

The keys for a record format are specified in DDS. When you use a Format 2 COPY statement, a table of comments is generated in the source program listing showing how the keys for the format are defined in DDS.
In addition, DDS keywords can be used to:

- Specify edit codes for a field (EDTCDE)
- Specify edit words for a field (EDTWRD)
- Specify that duplicate key values are not allowed for the file (UNIQUE)
- Specify a text description for a record format or a field (TEXT).

See the DDS Reference for a complete list of the DDS keywords that are valid for a database file.

---

**Figure 32. Example of a Field Reference File**

This example of a field reference file shows the definitions of the fields that are used by the CUSMSTL (customer master logical) file, which is shown in Figure 33 on page 109. The field reference file normally contains the definitions of fields that
are used by other files. The following text describes some of the entries for this field reference file.

1. The BASDAT field is edited by the Y edit code, as indicated by the keyword EDTCDE (Y). If this field is used in an externally described output file for a COBOL program, the COBOL-generated field is compatible with the data type specified in the DDS. The field is edited when the record is written. When the field is used in a program-described output file, compatibility with the DDS fields in the file is the user's responsibility. When DDS is not used to create the file, appropriate editing of the field in the COBOL program is also the user's responsibility.

2. The CHECK(MF) entry specifies that the field is a mandatory fill field when it is entered from a display work station. Mandatory fill means that all characters for the field must be entered from the display work station.

3. The ADDR and CITY fields share the same attributes that are specified for the NAME field, as indicated by the REFFLD keyword.

4. The RANGE keyword, which is specified for the CUSTYP field, ensures that the only valid numbers that can be entered into this field from a display work station are 1 through 5.

5. The COLHDG keyword provides a column head for the field if it is used by the Application Development Tools (Appl Dev Tools).

6. The ARBAL field is edited by the J edit code, as indicated by the keyword EDTCDE(J).

7. A text description (TEXT keyword) is provided for some fields. The TEXT keyword is used for documentation purposes and appears in various listings.

COBOL Specifications for Files Described Externally Using DDS

You can incorporate the file description in your program by coding a Format 2 COPY statement. The information from the external description is then retrieved by the COBOL compiler, and a COBOL data structure is generated.

The following pages provide examples of DDS usage and the COBOL code that would result from the use of a Format 2 COPY statement. (See “Format 2 COPY Statement (DD, DDR, DDS, or DDSR Option)” on page 112 for a detailed description of the Format 2 COPY statement.)

- Figure 33 on page 109 shows the DDS for a logical file and Figure 34 on page 110 shows the COBOL code generated.

- Figure 35 on page 111 describes the same file but includes the ALIAS keyword, and Figure 36 on page 112 shows the COBOL code generated.

Actual file processing within the Procedure Division is the same for both program-described and externally described files.
A logical file for processing the customer master physical file (CUSMSTP) is defined and named CUSMSTL.

The UNIQUE keyword indicates that duplicate key values for this file are not allowed.

One record format (CUSREC) is defined for the CUSMSTL file, which is to be based upon the physical file CUSMSTP.

The CUST field is identified as the key field for this file.

If field attributes (such as length, data type, and decimal positions) are not specified in the DDS for a logical file, the attributes are obtained from the corresponding field in the physical file. Any field attributes specified in the DDS for the logical file override the attributes for the corresponding field in the physical file.
the physical file. The definition of the fields in the physical file could refer to a field reference file. A field reference file is a data description file consisting of field names and their definitions, such as size and type. When a field reference file is used, the same fields that are used in multiple record formats have to be defined only once in the field reference file. For more information on a field reference file, see the Database Guide.

Figure 32 on page 107 shows an example of a field reference file that defines the attributes of the fields used in the database file. See the Database Guide for more information regarding field reference files.

```
01 CUS-MASTER.
   COPY DDS-CUSREC OF CUSLIB-CUSTMAST.
   *I-O FORMAT: CUSREC FROM FILE CUSTMAST OF LIBRARY CUSLIB CUSREC
   * CUSTOMER MASTER RECORD CUSREC
   *THE KEY DEFINITIONS FOR THE RECORD FORMAT CUSREC CUSREC
   *NUMBER NAME RETRIEVAL TYPE ALTSEQ CUSREC
   *0001 CUST ASCENDING AN NO CUSREC
   05 CUSREC.
      06 CUST PIC X(5). CUSREC
      * CUSTOMER NUMBER CUSREC
      06 NAME PIC X(20). CUSREC
      * CUSTOMER NAME CUSREC
      06 ADDR PIC X(20). CUSREC
      * CUSTOMER ADDRESS CUSREC
      06 CITY PIC X(20). CUSREC
      * CUSTOMER CITY CUSREC
      06 STATE PIC X(2). CUSREC
      * STATE ABBREVIATION CUSREC
      06 ZIP PIC S9(5) COMP-3. CUSREC
      * ZIP CODE CUSREC
      06 SHRCOD PIC X(6). CUSREC
      * CUSTOMER NAME SEARCH CODE CUSREC
      06 CUSTYP PIC 9(1). CUSREC
      * CUSTOMER TYPE CUSREC
      06 ARBAL PIC S9(6)V9(2) COMP-3. CUSREC
      * ACCT/REC BALANCE CUSREC
```

Figure 34. Example of the Results of the Format 2 COPY Statement (DDS)
This is the name associated with the ALIAS keyword, which will be included in the program. Available through the DDS ALIAS option, an alias is an alternative name that allows a data name of up to 30 characters to be included in a COBOL/400 program.
Format 2 COPY Statement (DD, DDR, DDS, or DDSR Option)

For general information about both formats of the COPY statement, see the COBOL/400 Reference.
You can use the Format 2 COPY statement (DD, DDR, DDS, or DDSR option) to create COBOL Data Division statements to describe a file that exists on the system. These descriptions are based on the version of the file in existence at compilation time. They do not make use of any DDS source statements for the file. Refer to the “COPY Statement” section of the COBOL/400 Reference for more information about the COPY statement.

**Note:** The Format 2 COPY statement (DD, DDR, DDS, or DDSR option) will be denoted by the term Format 2 COPY statement throughout this manual.

The Format 2 COPY statement can be used only in the Data Division. You should ensure that a group level item that has a level-number less than 05 precedes the statement.

The DD option is used to reference ALIAS (alternative) names. The specification of an ALIAS name in DDS allows a data name of up to 30 characters to be included in the COBOL program.

When the DD option is used, any ALIAS names present replace the corresponding DDS field names. All underscores in the ALIAS names are translated into hyphens before any replacing occurs.

The DDR option does everything that the DD option does. It also copies the internal DDS format field names, replacing the invalid COBOL characters @, #, $, and _ with the valid COBOL characters A, N, D, and - accordingly. This option also removes any underscores from the ends of the field names.

The DDS option copies in the internal DDS format field names. For examples of keys and key names that can be generated when you use the DDS option of the Format 2 COPY statement, see pages 121 through 127.

The DDSR option does everything that the DDS option does. It also copies the internal DDS format field names, replacing the invalid COBOL characters @, #, $, and _ with the valid COBOL characters A, N, D, and - accordingly. This option also removes any underscores from the ends of the field names.

The following shows the effect of the DDR or DDSR option on invalid COBOL field names:

<table>
<thead>
<tr>
<th>Original Field Name</th>
<th>Modified Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_A</td>
<td>FLD-A</td>
</tr>
<tr>
<td>NUMBER#1</td>
<td>NUMBERN1</td>
</tr>
<tr>
<td>POINT@7</td>
<td>POINTA7</td>
</tr>
<tr>
<td>BALANCE$</td>
<td>BALANCED</td>
</tr>
</tbody>
</table>

When the RECORD KEY clause specifies EXTERNALLY-DESCRIBED-KEY, a format can be copied only once under an FD. For example, if all of the formats of a file are copied under an FD, no other Format 2 COPY statement can be specified for the same file under that FD.

The format-name is the name of the DDS record format definition that is to be translated into COBOL data description entries. The format-name must follow the rules for formation of any COBOL/400 name.
If neither -I nor -O is specified, -I-O is assumed.

If format-name is specified without the Indicator attribute, and both -I and -O formats are to be generated, each record format is generated as a redefinition of an 05 elementary item defined as:

- The size of the largest record format that will be generated.

If ALL-FORMATS is specified (without the Indicator attribute) each record format is generated as a redefinition of an 05 elementary item defined as either:

- The size of the largest record format in the file, if the COPY statement appears in the File Section
- The size of the largest record format that will be generated, if the COPY statement appears outside of the File Section.

When the Indicator attribute is specified, no redefinition takes place. Instead, each of the formats generates a separate data structure.

More information can be found about the Indicator attribute in the section, "Indicator Attribute of the Format 2 COPY Statement" on page 118.

Library-name is optional. If it is not specified, the current job library list is used as the default value.

File-name is the name of an AS/400 system file. The generated DDS entries represent the record format defined in the file. The file must be created before the program is compiled.

If the file is a database file, a single I/O format is generated.

For all other file types, the description generated varies as follows:

- If -I is specified, the generated data description entries contain either:
  - The input and input/output fields for a nonsubfile format
  - The input, output, and input/output fields for a subfile format.
- If -O is specified, the generated data description entries contain either:
  - The output and input/output fields for a nonsubfile format
  - The input, output, and input/output fields for a subfile format.

**Note:** Subfile records with only output or input/output fields, and no field indicators specified, generate I/O formats.

If a separate storage area is needed in WORKING-STORAGE for each format, an individual COPY statement must be specified for each format.
For example, if you assume that the file CUSTMASTER contains two formats CUSADR and CUSTDETL, the following COPY statements could be specified:

```
SELECT FILE-X
  ASSIGN TO DATABASE-CUSTMASTER.

FD FILE-X
  LABEL RECORDS ARE STANDARD.
01 FILE-X-RECS.
    COPY DDS-ALL-FORMATS OF
      CUSTMASTER-QGPL. (See Note 1.)

WORKING-STORAGE SECTION.
01 ADR-REC.
    COPY DDS-CUSTADR OF
      CUSTMASTER. (See Note 2.)
01 DETAIL-REC.
    COPY DDS-CUSTDETL OF
      CUSTMASTER. (See Note 2.)
```

Notes:
1. This COPY statement generates only one storage area for all formats.
2. These COPY statements generate separate storage areas.

Indicators

Indicators are Boolean data items that can have the values B"0" or B"1".

When you define a record format for a file using DDS, you can condition the options using indicators; indicators can also be used to reflect particular responses. These indicators are known as OPTION and RESPONSE, respectively. Option indicators provide options such as spacing, underlining, and allowing or requesting data transfer from a program to a printer or display device. Response indicators provide response information to a program from a device, such as function keys pressed by a work station user, and whether data has been entered.

Indicators can be used with TRANSACTION files and FORMATFILE files, but never with database files.

Data Structures Generated

Different DDS keywords influence the creation of various types of data structures.

Format (Record) Level Structures

At the beginning of each format, a table of comments is generated in the source program listing. These comments provide details of the files used during compilation of the program. If there are record keys for the file, comments are also generated to show how the keys are defined in DDS. The record key entries that may appear in the table and the table heading are listed below.
<table>
<thead>
<tr>
<th>Heading</th>
<th>Possible Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER</td>
<td>key field number</td>
</tr>
<tr>
<td>NAME</td>
<td>key field name</td>
</tr>
<tr>
<td>RETRIEVAL</td>
<td>ASCENDING, DESCENDING</td>
</tr>
<tr>
<td>TYPE</td>
<td>ZONE, DIGIT, SIGNED, ABSVAL, AN (alphanumeric), N (numeric)</td>
</tr>
<tr>
<td></td>
<td>J (DBCS item), DDS - L (date), DDS - T (time), DDS - Z (timestamp), DDS - G (fixed-length graphic), VARLEN (variable-length character or bracketed DBCS item), G VARLEN (variable-length DBCS-graphic)</td>
</tr>
<tr>
<td>ALTSEQ</td>
<td>NO, YES</td>
</tr>
</tbody>
</table>

If redefinition is required to allow for the generation of multiple formats, a group level name is generated as follows:

```
05 file-name-RECORD
   PIC X(size of largest record).
```

For each format, a group level name is assigned as follows:

- **INPUT**
  05 format-name-I

- **OUTPUT**
  05 format-name-O

- **I/O Format**
  05 format-name

**Data Field Structures**

Field names, PICTURE definitions, and numeric usage clauses are derived directly from the internal DDS format field names (or ALIAS names in the case of the DD or DDR option) and data type representations. Field names and PICTURE definitions are constructed as follows:

```
06 field-name PIC
```

**Note:** See Figure 38 on page 117 for the appropriate COBOL PICTURE definition.
<table>
<thead>
<tr>
<th>Data Type (pos. 35)</th>
<th>DDS</th>
<th>Formats</th>
<th>If DDS pos. 36 &amp; 37 are blank</th>
<th>If DDS pos. 36 &amp; 37 are not blank</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHYSICAL, LOGICAL, PRINTER, AND COMMUNICATIONS FILES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b(Blank)</td>
<td>Default</td>
<td>PIC X(n)</td>
<td>PIC S9(n-m)V9(m)</td>
<td>PIC S9(n-m)V9(m)</td>
</tr>
<tr>
<td>P</td>
<td>Packed decimal</td>
<td>PIC S9(n) COMP-3</td>
<td>PIC S9(n-m)V9(m) COMP-3</td>
<td>PIC S9(n-m)V9(m) COMP-3</td>
</tr>
<tr>
<td>S</td>
<td>Zoned decimal/signed numeric</td>
<td>PIC S9(n) COMP-4</td>
<td>PIC S9(n-m)V9(m) COMP-4</td>
<td>PIC S9(n-m)V9(m) COMP-4</td>
</tr>
<tr>
<td>B</td>
<td>Binary</td>
<td>PIC S9(n) COMP-4</td>
<td>PIC S9(n-m)V9(m) COMP-4</td>
<td>PIC S9(n-m)V9(m) COMP-4</td>
</tr>
<tr>
<td>F</td>
<td>Floating point</td>
<td>PIC 9(5) COMP-4</td>
<td>PIC 9(5) COMP-4</td>
<td>PIC 9(5) COMP-4</td>
</tr>
<tr>
<td></td>
<td>single precision</td>
<td>PIC 9(10) COMP-4</td>
<td>PIC 9(10) COMP-4</td>
<td>PIC 9(10) COMP-4</td>
</tr>
<tr>
<td></td>
<td>double precision</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>A</td>
<td>Character</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>H</td>
<td>Hexadecimal data</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>L</td>
<td>Date</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>T</td>
<td>Time</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>Z</td>
<td>Timestamp</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>J</td>
<td>DBCS-Only data</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>E</td>
<td>DBCS-Either data</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>O</td>
<td>DBCS-Open data</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>G</td>
<td>DBCS-Graphic data</td>
<td>PIC X(2n)</td>
<td>PIC X(2n)</td>
<td>PIC X(2n)</td>
</tr>
<tr>
<td>DISPLAY FILES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b(Blank)</td>
<td>Default</td>
<td>PIC X(n)</td>
<td>PIC S9(n-m)V9(m)</td>
<td>PIC S9(n-m)V9(m)</td>
</tr>
<tr>
<td>X</td>
<td>Alphabetic Only</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>N</td>
<td>Numeric Shift</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>Y</td>
<td>Numeric Only</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>I</td>
<td>Inhibit Keyboard entry</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>W</td>
<td>Katakana</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>A</td>
<td>Alphanumeric Shift</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>D</td>
<td>Digits only</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>F</td>
<td>Floating point</td>
<td>PIC 9(5) COMP-4</td>
<td>PIC 9(5) COMP-4</td>
<td>PIC 9(5) COMP-4</td>
</tr>
<tr>
<td></td>
<td>single precision</td>
<td>PIC 9(10) COMP-4</td>
<td>PIC 9(10) COMP-4</td>
<td>PIC 9(10) COMP-4</td>
</tr>
<tr>
<td></td>
<td>double precision</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>M</td>
<td>Numeric-only character</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>S</td>
<td>Signed-numeric shift</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>E</td>
<td>DBCS-either</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>J</td>
<td>DBCS-only</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>O</td>
<td>DBCS-open</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>G</td>
<td>DBCS-graphic</td>
<td>PIC X(2n)</td>
<td>PIC X(2n)</td>
<td>PIC X(2n)</td>
</tr>
</tbody>
</table>

1 COBOL treats floating point fields as FILLER. See 'Floating Point Fields'.
2 In DDS, if the field has an attribute of VARLEN, the result is two additional bytes at the beginning of the field.
3 FILLER items by default. See 'Date, Time, and Timestamp Fields'.

Figure 38. Data Field Structures

Indicator Structures
If indicators are requested, and exist in the format, an additional group name (06 level) is generated at the beginning of the structure, followed by entries (07 level) for the relevant individual indicators.

06 format-name(-I or -O)-INDIC.
07 INxx PIC 1 INDIC xx.

where xx is the indicator number.
For example:

```
06 SAMPLE1-I-INDIC.
   07 IN01 PIC 1 INDIC 01.
   07 IN04 PIC 1 INDIC 04.
   07 IN05 PIC 1 INDIC 05.
   07 IN07 PIC 1 INDIC 07.
06 FLD1 PIC ... .
06 FLD2 PIC ... .
```

**Indicator Attribute of the Format 2 COPY Statement**

The Indicator attribute specifies if data description entries are generated for indicators.

If the Indicator attribute is specified, data description entries are generated for indicators, but not for data fields. A 05 group level entry is generated as follows:

- If the COPY is for a single structure (for example, COPY DDS-format-name-INDIC)
  
  05 format-name-I. (or -O as appropriate)

- If the COPY is for multiple structures (for example, COPY DDS-ALL-FORMATS-INDIC)
  
  05 file-name-RECORD.

The data description entries that are generated are determined by which one of the usage attributes (I, O, or I-O) is specified or assumed in the COPY statement.

- If ...I-INDICATOR... is specified, data description entries for input (response) indicators are generated for indicators used in the input record area.
- If ...O-INDICATOR... is specified, data description entries for output (option) indicators are generated for indicators used in the output record area.
- If ...I-O-INDICATOR... is specified or assumed, separate data description entries for both input and output (response and option) indicators are generated for indicators used in the input and output record areas.

If the Indicator attribute is not specified, generation of data description entries for indicators depends on if the file had the keyword INDARA specified in the DDS at the time it was created.

- If INDARA was not specified, data description entries are generated for both data fields and indicators.
- If INDARA was specified, data description entries are generated for data fields only, not for indicators.

**Generation of I/O Formats**

When all field descriptions are identical, and you have requested INPUT or OUTPUT fields implicitly or explicitly, only one set of field descriptions is generated. This type of description is annotated with a comment line reading, “I-O FORMAT: format-name”. Neither -I nor -O is appended to the record format name.

**Note:** This always happens for database files because all field descriptions within a database file are identical.
For example:

```
01 RCUSREC.
   COPY DDS-CUSREC-I OF CUSFILE.
   * I-O FORMAT: CUSREC FROM FILE CUSFILE OF LIBRARY CUSLIB CUSREC
   * THE KEY DEFINITIONS FOR RECORD FORMAT CUSREC
   * NUMBER NAME RETRIEVAL TYPE ALTSEQ
   * 0001 ARBAL ASCENDING SIGNED NO
   * 0002 AREACD DESCENDING ABSVAL NO
   05 CUSREC.
   06 ARBAL PIC S9(7)V9(2) COMP-3 CUSREC
   06 AREACD PIC S9(3) COMP-3. CUSREC
   06 BOSTAZ PIC X(1). CUSREC
   06 CNTCT PIC X(15). CUSREC
   06 CRCHKZ PIC S9(2). CUSREC
   06 CSTAT PIC X(1). CUSREC
   06 CUSTNZ PIC S9(6). CUSREC
   06 DLORD PIC S9(6). CUSREC
   06 DSCPCZ PIC S9(2)V9(3) COMP-3. CUSREC
   06 INDUS PIC S9(2). CUSREC
   06 NAME1 PIC X(25). CUSREC
   06 NAME2 PIC X(25). CUSREC
   06 NAME3 PIC X(25). CUSREC
   06 NAME4 PIC X(25). CUSREC
   06 PHONE PIC S9(7) COMP-3. CUSREC
   06 PRICIZ PIC S9(2). CUSREC
   06 SHPINZ PIC X(25). CUSREC
   06 SLSMAZ PIC X(3). CUSREC
   06 TAXCDZ PIC S9(2). CUSREC
   06 TERMSZ PIC S9(2). CUSREC
```

**Figure 39. Example of Copy DDS Showing I/O Formats**

**Redefinition of Formats**

Pay particular attention to the REDEFINES clause that may be generated for the ALL-FORMATS or -I-O phrases. Because all formats are redefined on the same area (generally a buffer area), several field names can describe the same area of storage, and unpredictable results can occur if the entire format area is not reinitialized prior to each output operation.

Data items that are subordinate to the data item specified in a MOVE CORRESPONDING statement do not correspond and are not moved when they contain a REDEFINES clause or are subordinate to a redefining item.

To avoid reinitialization, multiple Format 2 COPY statements using -I and -O suffixes can be used to create separate areas of storage in the Working-Storage section for each format or format type (input or output). READ INTO and WRITE FROM statements can be used with these record formats.
For example:

```
FD ORDER-ENTRY-SCREEN ...
  01 ORDER-ENTRY-RECORD ...
  :
  WORKING-STORAGE SECTION.
  01 ORDSFL-I-FORMAT.
       COPY DDS-ORDSFL-I OF DOESCR.
  01 ORDSFL-O-FORMAT.
       COPY DDS-ORDSFL-O OF DOESCR.
  :
  PROCEDURE DIVISION.
  :
  READ SUBFILE ORDER-ENTRY-SCREEN NEXT MODIFIED RECORD
       INTO ORDSFL-I-FORMAT FORMAT IS "ORDSFL"
       AT END SET NO-MODIFIED-SUBFILE-RCD TO TRUE.
  :
  MOVE CORR ORDSFL-I TO ORDSFL-O.
  REWRITE SUBFILE ORDER-ENTRY-RECORD FROM ORDSFL-O-FORMAT
       FORMAT IS "ORDSFL" ...
  :
```
Figure 40. Data Description Specifications for a Physical File

The physical file described by Figure 40 forms a basis for the examples that follow. Each example refers to a logical file (derived from the physical file) that specifies EXTERNALLY-DESCRIBED-KEY in its SELECT clause.
Example Using CONCAT Keyword

For the logical file described by Figure 41, COPY DDS generates keys and key names derived from the physical file:
FD LF1 LABEL RECORDS ARE STANDARD.
01 LOG-RECORD.
   COPY DDS-ALL-FORMATS OF LF1.
   05 LF1-RECORD PIC X(8).
   * I-O FORMAT:RECORD1 FROM FILE LF1 OF LIBRARY COPYDDS
   *
   *THE KEY DEFINITIONS FOR RECORD FORMAT RECORD1
   * NUMBER NAME RETRIEVAL TYPE ALTSEQ
   * 0001 MTH-DDS ASCENDING AN NO
   * KEY NAME ORIGIANATES FROM PHYSICAL FILE
   * 0002 DAY-DDS-DDS ASCENDING AN NO
   * KEY NAME ORIGIANATES FROM PHYSICAL FILE
   05 RECORD1 REDEFINES LF1-RECORD.
   06 DATE-DDS PIC X(8).
   06 FILLER REDEFINES DATE-DDS.
   07 MTH-DDS PIC X(2).
   07 DAY-DDS-DDS PIC X(2).
   07 FILLER PIC X(4).

Figure 42. Example Using the CONCAT Keyword

The COPY statement adds the suffix -DDS to the field names MTH and DATE because MTH is a key that originates from the physical file, and DATE is a COBOL reserved word. The COPY statement adds the suffix -DDS twice to the field name DAY because DAY is both a key that originates from the physical file and a COBOL reserved word.

Note that if you move your COPY statement from the File Section to the Working-Storage Section or to the Linkage Section, the fields subordinate to DATE-DDS are no longer available:

WORKING- STORAGE SECTION.
01 WRK-RECORD.
   COPY DDS-ALL-FORMATS OF LF1.
   05 LF1-RECORD PIC X(8).
   * I-O FORMAT:RECORD1 FROM FILE LF1 OF LIBRARY COPYDDS
   *
   05 RECORD1 REDEFINES LF1-RECORD.
   06 DATE-DDS PIC X(8).

Figure 43. Example Using the CONCAT Keyword-- Working-Storage Section
Example Using RENAME Keyword

**Figure 44. Data Description Specifications Using the RENAME Keyword**

For the logical file described by Figure 44, COPY DDS generates a key and key name derived from the physical file:

<table>
<thead>
<tr>
<th>Sequence Number</th>
<th>Form Type</th>
<th>And/Or/Comment (A/O/*)</th>
<th>Not (N)</th>
<th>Indicator</th>
<th>Type of Name of Spec (b/R/H/J/K/S/O)</th>
<th>Location</th>
<th>Usage (b/O/I/B/H/M/N/P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Reference (R)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Decimal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Positions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Function</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**A**

**LOGICAL FILE LF2 FOR RENAME KEYWORD EXAMPLES**

| A* | LOGICAL RECORD2 P FILE(PF1) |
| A* | LOGICAL MTH RENAME(MTH) |
| A  | LOGICAL K MTH |
FD LF2 LABEL RECORDS ARE STANDARD.
01 LOG-RECORD.
    COPY DDS-ALL-FORMATS OF LF2.
    05 LF2-RECORD PIC X(2).
* I-O FORMAT:RECORD2 FROM FILE LF2 OF LIBRARY COPYDDS
* + THE KEY DEFINITIONS FOR RECORD FORMAT RECORD2
  + NUMBER NAME RETRIEVAL TYPE ALTSEQ
  + 0001 MTH-DDS ASCENDING AN NO
  + KEY NAME ORIGINATES FROM PHYSICAL FILE
  05 RECORD2 REDEFINES LF2-RECORD.
   06 MONTH PIC X(2).
   06 MTH-DDS REDEFINES MONTH PIC X(2).

Figure 45. Using the RENAME Keyword

The COPY statement adds the suffix -DDS to the field name MTH because MTH is a key that originates from the physical file.
### Example Using SST Keyword

#### Figure 46. Data Description Specifications Using the SST Keyword

<table>
<thead>
<tr>
<th>Condition Name</th>
<th>Sequence Number</th>
<th>Condition Indicator</th>
<th>Condition Name</th>
<th>Length</th>
<th>Data Type</th>
<th>Data Type/Keyboard Shift</th>
<th>Usage</th>
<th>Comment (A/O/*)</th>
<th>Not (N)</th>
<th>Indicator</th>
<th>Indicator</th>
<th>Indicator</th>
</tr>
</thead>
<tbody>
<tr>
<td>LO GICAL F I LE LF3 FO R SST KEY W O R D EXA M PLES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A*</td>
<td>R RECORDS</td>
<td>P FILE(PF1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A*</td>
<td>YY</td>
<td>SST(YEAR 2 2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A*</td>
<td>K YY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
For the logical file described by Figure 46 on page 126, COPY DDS generates the following specifications:

```cobol
FD LF3 LABEL RECORDS ARE STANDARD.
01 LOG-RECORD.
   COPY DDS-ALL-FORMATS OF LF3.
   05 LF3-RECORD PIC X(2).
* I-O FORMAT:RECORD3 FROM FILE LF3 OF LIBRARY COPYDDS
* THE KEY DEFINITIONS FOR RECORD FORMAT RECORD3
  * NUMBER NAME RETRIEVAL TYPE ALTSEQ
  * 0001 YY ASCENDING AN NO
  05 RECORD3 REDEFINES LF3-RECORD.
  06 YY PIC X(2).
```

Figure 47. Using the SST Keyword

The COPY statement does not add a suffix to the field name YY because YY is neither a key that originates from the physical file nor a COBOL reserved word.

Additional Notes on Field and Format Names
If the generated field name is a COBOL reserved word, the suffix -DDS is added to the field name.

The REPLACING phrase cannot be used to change the name of a key field when EXTERNALLY-DESCRIBED-KEY is used.

Floating-Point Fields
COBOL treats floating-point fields as FILLER. The fields can contain floating-point values set outside of COBOL. A COMP-4 definition is generated to maintain proper alignment in the record, but the data is not in binary format. No attempt must be made to use floating-point data for processing in the COBOL program.

Floating-point key fields are not allowed. In cases where some formats exist with a floating-point key field and other formats do not, use one or more Format 2 COPY statements with specific format names, rather than the ALL-FORMATS option.

Note: If you have not specified your own program collating sequence, you can create a record containing floating-point fields in your COBOL program by moving LOW-VALUES to the entire record before moving in the values of the non-floating-point fields. This will give the floating-point fields in the record a value of zero. Note that the above method is only recommended if valid floating-point fields with a value of zero are desirable for your particular application.

REPLACING Phrase in Format 2 COPY Statement
The REPLACING phrase can be used to replace any of the generated COBOL source, including the level numbers and the format-name. Note the following exception:
- When RECORD KEY IS EXTERNALLY-DESCRIBED-KEY is specified, the REPLACING phrase cannot change the name of a field that is a key.
Access Path

The description of an externally described file contains the access path that describes how records are to be retrieved from the file. Records can be retrieved based on an arrival sequence (nonkeyed) access path or on a keyed sequence access path.
The arrival sequence access path is based on the order in which the records are stored in the file. Records are added only to the end of the file.

For the keyed sequence access path, the sequence in which records are retrieved from the file is based on the contents of the key fields defined in the DDS for the file. For example, in the DDS shown in Figure 33 on page 109, CUST is defined as the key field. The keyed sequence access path is updated whenever records are added, deleted, or when the contents of a key field change.

See the Database Guide for a complete description of the access paths for an externally described database file.

**Record Keys and Common Keys**

For a keyed sequence access path, one or more fields can be defined in the DDS to be used as the key fields for a record format. All record types in a file do not have to have the same key fields. For example, an order header record can have the ORDER field defined as the key field, and the order detail records can have the ORDER and LINE fields defined as the key fields.

The key for a file is determined by the valid keys for the record types in that file. The file’s key is determined in the following manner:

- If all record types in a file have the same number of key fields defined in DDS that are identical in attributes, the key for the file consists of all fields in the key for the record types. (The corresponding fields do not have to have the same name.) For example, if the file has three record types and the key for each record type consists of fields A, B, and C, the file’s key consists of fields A, B, and C. That is, the file’s key is the same as the records’ key.

- If all record types in the file do not have the same key fields, the key for the file consists of the key fields common to all record types. For example, a file has three record types and the key fields are defined as follows:
  - REC1 contains key field A.
  - REC2 contains key fields A and B.
  - REC3 contains key fields A, B, and C.

  Then the file’s key is field A, the key field common to all record types.

- If no key field is common to all record types, any keyed reference to the file will always return the first record in the file.

In COBOL, you must specify a RECORD KEY for an indexed file to identify the record you want to process. COBOL compares the key value with the key of the file or record, and processes the specified operation on the record whose key matches the RECORD KEY value.

When RECORD KEY IS EXTERNALLY-DESCRIBED-KEY is specified:

- If the FORMAT phrase is specified, the compiler builds the search argument from the key fields in the record area for the specified format.

- If the FORMAT phrase is not specified, the compiler builds the search argument from the key fields in the record area for the first record format defined in the program for that file.

**Note:** For a file containing multiple key fields to be processed in COBOL, the key fields must be contiguous in the record format used by the COBOL
program, except when RECORD KEY IS EXTERNALLY-DESCRIBED-KEY is specified.

Overriding or Adding COBOL Functions to the External Description
In addition to placing the external file description in the program through the use of the Format 2 COPY statement, you can also use standard record definition and redefinition to describe external files or to provide a group definition for a series of fields. It is the programmer’s responsibility to ensure that program-described definitions are compatible with the external definitions of the file.

Level Checking
When a COBOL/400 program uses an externally described file, the operating system provides a level check function (LVLCHK). This function ensures that the format has not changed since compilation time.

The compiler always provides the information required by level checking when an externally described file is used (that is, when a record description was defined for the file by using the Format 2 COPY statement). Only those formats that were copied by the Format 2 COPY statement under the FD for a file are level checked. The level check function will be initiated at run time based on the selection made on the create, change, or override file commands. The default on the create file command is to request level checking. If level checking was requested, level checking occurs on a record format basis when the file is opened. If a level check error occurs, COBOL sets a file status of 39 at OPEN time.

When no level checking was requested, and the file is re-created using an existing format, existing COBOL programs that use that format may not work without recompilation, depending on the changes to the format. For instance,

- A change of keys will certainly cause a failure of the program on any I/O statement
- A change in the record length will cause any REWRITE to fail
- A change in the record layout can cause various errors in the processing of such a record.

You should use extreme caution when using COBOL programs without level checking or recompiling the programs.

Note: The compiler does not provide level checking for program-described files.

For more information on level checking, see the Data Management Guide.

Declaring Data Items Using CVTOPT Data Types
The COBOL/400 compiler allows you to convert variable-length fields from externally described files and SAA database data types to standard COBOL data items. The SAA data types you can convert are date, time, timestamp, and DBCS-graphic. COBOL/400 provides limited support for these data types.
Variable-length Fields

You can bring a variable-length field into your program if you specify *VARCHAR on the CVTOPT parameter of the CRTCBLPGM command, or the VARCHAR option of the PROCESS statement. When *VARCHAR is specified, your COBOL/400 program will convert a variable-length field from an externally described file into a COBOL/400 group item.

An example of such a group item is:

```
06 ITEM1.
   49 ITEM1-LENGTH   PIC S9(4) COMP-4.
   49 ITEM1-DATA     PIC X(n).
```

where n represents the maximum length of the variable-length field. Within the program, the PIC S9(4) COMP-4 is treated like any other declaration of this type, and the PIC X(n) is treated as standard alphanumeric.

Since the maximum value that ITEM1-LENGTH can hold is 9 999, this is the length of the longest variable-length field you can write from a COBOL program.

When *VARCHAR is not specified, variable-length fields are ignored and declared as FILLER fields in COBOL/400 programs. If *NOVARCHAR is specified, the item is declared as follows:

```
06 FILLER   PIC x(n+2).
```

For syntax information, see the CVTOPT parameter on page 23.

Your program can perform any valid character operations on the generated data portion; however, because of the structure of the field, the length portion must be valid binary data. This data is not valid if it is negative, or greater than the maximum field length.

If the first two bytes of the field do not contain a valid binary number, an error will occur if you try to WRITE or REWRITE a record containing the field (or UPDATE or PUT the field in a database), and file status 90 is returned.

The following conditions apply when you specify variable-length fields:

- If a variable-length field is encountered when a field is extracted for an externally described file or an externally described data structure, it is declared in a COBOL/400 program as a fixed-length character field.
- For single-byte character fields, the length of the declared COBOL/400 field is the length of the DDS field plus 2 bytes.
- For DBCS-graphic data fields, the length of the declared COBOL/400 field is two times the length of the DDS field plus 2 bytes. For more information on graphic data types, see “DBCS-Graphic Fields” on page 133. The two extra bytes in the COBOL/400 field contain a binary number that represents the current length of the variable-length field. Figure 50 on page 132 shows the COBOL/400 field length of variable-length fields.
Your COBOL/400 program can perform any valid character calculation operations on the declared fixed-length field. However, because of the structure of the field, the first two bytes of the field must contain valid binary data (invalid current field-length data is non-numeric, less than 0, or greater than the DDS field length.) An error occurs for an input or output operation if the first two bytes of the field contain invalid field-length data; file status 90 is returned.

- If you do not specify *VARCHAR, you can encounter problems performing WRITE operations on variable-length fields, because you cannot assign a value to FILLER. The two-byte field may have a value (for example 'X'4040') which gives a length beyond the range allowed for the field. This causes an I/O error.

To see an example of a program using variable-length fields, refer to “Examples” on page 134.

**Date, Time, and Timestamp Fields**

Date, time, and timestamp fields are brought into your program only if you specify the *DATETIME option of the CRTCBPLGM CVTOPT parameter, or the DATETIME option of the PROCESS statement. For a description and the syntax of the CVTOPT parameter, see page 23. If *DATETIME is not specified, date, time, and timestamp fields are ignored and are declared as FILLER fields in your COBOL/400 program.

Date, time or timestamp fields are brought into a COBOL/400 program as fixed-length character fields. Your COBOL/400 program can perform any valid character operations on the fixed-length fields. These operations will follow the standard COBOL rules for alphanumeric data items.

The date, time, and timestamp data types each have their own format.

If a field containing date, time, or timestamp information is updated by your program, and the updated information is to be passed back to your database, the format of the field must be exactly the same as it was when the field was retrieved from the database. If you do not use the same format, an error will occur. For information on valid formats for each data type, see the DDS Reference.

If you try to WRITE a record before moving an appropriate value to a date, time, or timestamp field, the WRITE operation will fail, and file status 90 will be returned.
If you declare date, time or timestamp items in your program as FILLER, do not attempt to WRITE records containing these fields, since you will not be able to set them to values that will be accepted by the system.

**Null-capable Fields**

Although your program can process null-capable fields, null values are not supported. READ, SORT, and MERGE operations can be performed on null-capable fields, but if the fields actually contain null values, errors occur.

**DBCS-Graphic Fields**

The DBCS-graphic data type is a character string in which each character is represented by 2 bytes. The DBCS-graphic data type does not contain shift-out (SO) or shift-in (SI) characters. The difference between single-byte and DBCS-graphic data is shown in the following figure:

```
<table>
<thead>
<tr>
<th>1 byte</th>
<th>1 byte</th>
<th>1 byte</th>
<th>1 byte</th>
<th>Single-byte data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 char 1 char 1 char</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>1 byte</th>
<th>1 byte</th>
<th>1 byte</th>
<th>1 byte</th>
<th>DBCS-graphic data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 character 1 character</td>
</tr>
</tbody>
</table>
```

*Figure 51. Comparing Single-byte and Graphic Data*

DBCS-graphic data is brought into your COBOL/400 program only if you specify the *GRAPHIC value on the CVTOPT parameter of the CRTCBLPGM command, or the CVTGRAPHIC option of the PROCESS statement. If you do not specify DBCS-graphic data, graphic data is ignored and declared as FILLER fields in your COBOL/400 program. For a description and the syntax of the CVTOPT parameter, see page 23.

The following conditions apply when DBCS-graphic data is specified:

- DBCS-graphic data is copied into a COBOL/400 program as a fixed-length alphanumeric field.
- Every DBCS-graphic data character has a length of 2 bytes.
- Every fixed-length DBCS-graphic data field has a length of 2 bytes times the number of characters in the field. For a description of the field length of variable-length graphic data fields, see “Variable-length Fields” on page 131.
- Your COBOL/400 program can perform any valid character operations on the fixed-length fields.
Variable-length DBCS-Graphic Fields

You can use variable-length fields in combination with DBCS-graphic data types, to specify variable-length DBCS-graphic data. To specify variable-length DBCS-graphic data, specify +VARCHAR and +GRAPHIC for the CVTOPT parameter of the CRTCLBLPGM command, or the VARCHAR and CVTGRAPHIC options for the PROCESS statement.

If you specify either of the following: CVTOPT(+NOVARCHAR +NOGRAPHIC) or CVTOPT(+NOVARCHAR +GRAPHIC) and the compiler encounters a variable-length DBCS-graphic data item, the resulting program contains the following:

```cobol
06 FILLER PIC X(2n+2).
   * (Variable-length field)
```

where \( n \) is the number of characters in the DDS field.

If you specify CVTOPT(+VARCHAR +NOGRAPHIC), and the compiler encounters a variable-length DBCS-graphic data item, the resulting program contains the following:

```cobol
06 NAME.
   * (Variable-length field)
   49 NAME-LENGTH PIC S9(4) COMP-4.
   * (Number of 2-byte characters)
   49 FILLER PIC X(2n).
   * (Graphic field)
```

where \( n \) is the number of characters in the DDS field.

If you specify CVTOPT(+VARCHAR +GRAPHIC), and the compiler encounters a variable-length DBCS-graphic data item, the resulting program contains the following:

```cobol
06 NAME.
   * (Variable-length field)
   49 NAME-LENGTH PIC S9(4) COMP-4.
   * (Number of 2-byte characters)
   49 NAME-DATA PIC X(2n).
   * (Graphic field)
```

where \( n \) is the number of characters in the DDS field.

Examples

Figure 52 on page 135 shows an example of a DDS file that defines a variable-length DBCS-graphic data item. Figure 53 on page 136 shows the COBOL/400 program using a COPY DDS statement, and the resulting listing when the program is compiled.
Figure 52. DDS File Defining a Variable-Length Graphic Data Field
Figure 53. COBOL/400 Program Using Variable-Length DBCS-Graphic Data Items
Cross-system Data Considerations

Coded character set identifiers (CCSIDs) can help you to maintain the integrity of character data across systems.

Character Data Representation Architecture (CDRA) defines CCSID values to identify the code points used to represent characters, and to convert these codes as needed to preserve their meanings.

As a consequence of CDRA conversion, you might have substitution characters (X'3F') in your data. If you write these characters to a display, the results will not be predictable.

For more information about CCSIDs and CDRA, see System Operation, SC41-3203 and the Data Management Guide.
Chapter 8. Transaction Files

IBM Extension

This chapter describes the COBOL/400 language extensions that support work
stations and program-to-program communication.

The TRANSACTION file organization allows a COBOL program to communicate
interactively with:

- One or more work station users
- One or more programs on a remote system
- One or more devices on a remote system.

The AS/400 system permits you to communicate with a program or device (such as
Asynchronous communication types) on a remote system. For a detailed dis-
cussion of these devices, see the ICF Programmer’s Guide.

Program-Described Transaction Files

COBOL TRANSACTION files are usually externally described. If these files are
program-described, only simple display formatting can be performed. All field-level
descriptions are defined in the COBOL program.

Do not send internal (packed) or binary data (COMP, COMP-3, or COMP-4) to a
display station as output data. Such data can contain display station control char-
acters that can cause unpredictable results.

See the Data Management Guide for more information about using program-
described display files.

Externally Described Transaction Files

A COBOL TRANSACTION file uses an externally described file that contains file
information and a description of the fields in the records. The records in this file
can be described to the COBOL program by the Format 2 COPY statement.

The Format 2 COPY Statement

Format 2 COPY statements are used to generate COBOL Data Division statements
within source programs to describe files that exist on the system.

Note: The term Format 2 COPY statement is used throughout this manual to
describe the COPY statement (DD, DDR, DDS, or DDSR option).

For more information about the Format 2 COPY statement, see “Format 2 COPY
Statement (DD, DDR, DDS, or DDSR Option)” on page 112.
Data Description Specifications

Data description specifications (DDS) are a description of the user’s database or device files that are entered into the system in a fixed form. The description is then used to create files.

In addition to the field descriptions (such as field names and attributes), the data description specifications (DDS) for a display device file:

- Specify the line number and position number entries for each field and constant to format the placement of the record on the display.
- Specify attention functions such as underlining and highlighting fields, reverse image, or a blinking cursor.
- Specify validity checking for data entered at the display work station. Validity checking functions include:
  - Detecting fields where data is required
  - Detecting mandatory fill fields
  - Detecting incorrect data types
  - Detecting data for a specific range
  - Checking data for a valid entry
  - Performing modules 10 or 11 check digit verification.
- Control display management functions such as when fields are to be erased, overlaid, or retained when new data is displayed.
- Associate indicators 01 through 99 with function keys designated as type CA or CF. If a function key is designated as CF, both the modified data record and the response indicator are returned to the program. If a function key is designated as CA, the response indicator is returned to the program, but the data record usually contains default values for input-only fields and values written to the format for hidden output/input fields. For more information about type CF and CA function keys, see the DDS Reference.
- Assign an edit code (EDTCDE keyword) or edit word (EDTWRD keyword) to a field to specify how the field’s values are to be displayed.
- Specify subfiles.

Display format data defines or describes a display. A display device record format contains three types of fields:

- **Input Fields**: Input fields pass from the device to the program when the program reads a record. Input fields can be initialized with a default value; if the default value is not changed, the default value passes to the program. Uninitialized input fields are displayed as blanks where the work station user can enter data.
- **Output Fields**: Output fields pass from the program to the device when the program writes a record to a display. The program or the record format in the device file can provide output fields.
- **Output/Input (both) Fields**: An output/input field is an output field that can be changed to become an input field. Output/input fields pass from the program when the program writes a record to a display and pass to the program when the program reads a record from the display. Output/input fields are used when the user is to change or update the data that is written to the display from the program.
For a detailed description of a data communications file, see the *ICF Programmer's Guide*. For more information on externally defined display files, see the *Data Management Guide*. For a list of the valid data description specifications (DDS) keywords, see the *DDS Reference*.

Figure 54 shows an example of the DDS for a display device file:

![Figure 54. Example of the Data Description Specifications for a Display Device File](image)

This display device file contains two record formats: CUSPMT and CUSFLDS.

1. The attributes for the fields in this file are defined in the CUSMSTP field reference file. For example, EDTCDH(J) is defined in CUSMSTP for the field ARBAL.

2. The F3 key is associated with indicator 15, with which the user ends the program.

3. The ERRMSG keyword identifies the error message that is displayed if indicator 99 is set on in the program that uses this record format.
The OVERLAY keyword is used for the record format CUSFLDS so that the CUSPMT record on the display will not be erased when the CUSFLDS record is written to the display.

The constants such as ‘Name’, ‘Address’, and ‘City’ describe the fields that are written out by the program.

The line and position entries identify where the fields or constants are written on the display.

Processing an Externally Described Transaction File

When an externally described TRANSACTION file is processed, the operating system transforms data from the program to the format specified for the file and displays the data. When data passes to the program, the data is transformed to the format used by the program.

The operating system provides device control information for performing input/output operations for the device. When an input record is requested from the device, the operating system issues the request, and then removes device control information from the data before passing the data to the program. In addition, the operating system can pass indicators to the program indicating which, if any, fields in the record have changed.

When the program requests an output operation, it passes the output record to the operating system. The operating system provides the necessary device control information to display the record. It also adds any constant information specified for the record format when the record is displayed.

When a record passes to a program, the fields are arranged in the order in which they are specified in the DDS. The order in which the fields are displayed is based on the display positions (line numbers and positions) assigned to the fields in the DDS. Therefore, the order in which the fields are specified in the DDS and the order in which they appear on the display need not be the same.

Using Indicators with Transaction Files

Indicators are Boolean data items that can have the values B"0" or B"1".

When you define a record format for a file using DDS, you can condition the options using indicators; indicators can also be used to reflect particular responses. These indicators are known as OPTION and RESPONSE, respectively.

Option indicators provide options such as spacing, underlining, and allowing or requesting data transfer from a program to a printer or display device. Response indicators provide response information to a program from a device, such as function keys pressed by a work station user, and whether data has been entered.

Indicators can be passed with data records in a record area, or outside the record area in a separate indicator area.
Indicators in a Separate Indicator Area

If you specify the file level keyword INDARA in the DDS, all indicators defined in the record format or formats for that file are passed to and from the program in a separate indicator area, not in the record area. For information on how to specify the INDARA keyword, see the DDS Reference.

The file control entry for a file that has INDARA specified in its DDS must have the separate indicator area attribute, SI, as part of the assignment-name.

The advantages of using a separate indicator area are as follows:

- The number and order of indicators used in an I/O statement for any record format in a file need not match the number and order of indicators specified in the DDS for that record format.
- The program associates the indicator number in a data description entry with the appropriate indicator.

Indicators in the Record Area

If the keyword INDARA is not used in the DDS of the file, indicators are created in the record area. When indicators are defined in a record format for a file, they are read, rewritten, and written with the data in the record area.

The number and order of indicators defined in the DDS for a record format for a file determines the number and order in which the data description entries for the indicators in the record format must be coded in the program.

The file control entry for a file that does not have the INDARA keyword specified in the DDS associated with it must not have the separate indicator area attribute, SI, as part of the assignment-name.

If a Format 2 COPY statement is used to copy indicators into a source program, the indicators are defined in the order in which they are specified in the DDS for the file.

ASSIGN Clause and the Separate Indicator Area Attribute

The rules for the ASSIGN clause are as follows:

- Device must be WORKSTATION
- If -SI is coded, file-name must refer to a file that has the file level keyword INDARA specified in its DDS.

For more information about the ASSIGN clause, see “ASSIGN Clause” on page 172.
Data Description Entry—Boolean Data

When you use indicators in a COBOL program, you must describe them as Boolean data items using the data description entry for Boolean data.

Special Considerations
The special considerations for the clauses used with the Boolean data follow. All other rules for clauses are the same as those for other data as described in the "COBOL Program Structure" section of the COBOL/400 Reference.

PICTURE Clause: An elementary Boolean data name is defined by a PICTURE containing a single 1.

USAGE Clause: USAGE must be defined implicitly or explicitly as DISPLAY.
**OCCURS Clause:** When the OCCURS clause and the INDICATOR clause are both specified at an elementary level, a table of Boolean data items is defined with each element in the table corresponding to an external indicator. The first element in the table corresponds to the indicator number specified in the INDICATOR clause; the second element corresponds to the indicator that sequentially follows the indicator specified by the INDICATOR clause.

For example, if the following is coded, SWITCHES (1) corresponds to indicator 16, SWITCHES (2) corresponds to indicator 17, ..., and SWITCHES (10) corresponds to indicator 25:

```
07 SWITCHES PIC 1
    OCCURS 10 TIMES
    INDICATOR 16.
```

**INDICATOR Clause:** If indicator fields are in a separate indicator area, the INDICATOR clause associates an indicator defined in DDS with a Boolean data item. If indicator fields are in the record area, the INDICATOR clause is syntax-checked, but is treated as a comment.

Integer-3 must have a value of 1 through 99.

The INDICATOR clause must be specified at an elementary level only.

**VALUE Clause:** The VALUE clause specifies the initial content of a Boolean data item. The allowable values for Boolean literals are B"0", B"1", and ZERO.

**LIKE Clause:** You cannot use this clause to change the length of the data item.

**INDICATORS Phrase**

When the INDICATORS phrase is used in READ, REWRITE, and WRITE statements (see Figure 57 on page 150), it specifies which indicators are to be read, rewritten, and written.

The identifier specified in the INDICATORS phrase can be either of the following:

- An elementary Boolean data item
- A group item with elementary Boolean data items subordinate to it. (The Boolean data items can be anywhere in the group, but they are the only items you can read, write, or rewrite.)

The identifier cannot be subordinate to an item that is subject to an OCCURS clause.

**Indicators in a Separate Indicator Area**

If INDARA is specified in the DDS for the file, the use of the indicators referenced in the INDICATORS phrase is based on indicator number.

- In a READ statement, only the response indicator numbers referenced by the INDICATORS phrase are updated. Indicators specified in the DDS for the format but not referenced by the INDICATORS phrase are ignored. Indicators referenced by the INDICATORS phrase but not specified in the DDS are not modified.
- In a WRITE or REWRITE statement, only the option indicators referenced by the INDICATORS phrase are used. Indicators specified in the DDS for the
format but not referenced by the INDICATORS phrase are assumed to be OFF.
Indicators referenced by the INDICATORS phrase but not used in the DDS for
the format are ignored.

If the INDICATORS phrase is not specified, the following occurs:

• In the READ statement, indicators are not updated.
• In a WRITE or REWRITE statement, indicators are treated as though they are
  set to OFF.

Indicators in the Record Area

If INDARA is not specified in the DDS for the file, the size of the identifier in the
INDICATORS phrase of an I/O statement (see Figure 57 on page 150) should be
equal to the number of option or response indicators defined in the DDS for that
format.

• In a READ statement, the identifier size should be equal to the number of
  response indicators.
• In a REWRITE or WRITE statement, the identifier size should be equal to the
  number of option indicators.

The contents of the identifier are not checked, but are copied to or from the begin-
ning of the record, on a byte-by-byte basis; indicator numbers are ignored.

If the INDICATORS phrase is omitted, the data in the indicator fields in the record
are still passed in the record area. The INDICATORS phrase is only used to copy
indicators into the record area before a WRITE or REWRITE statement, or out of
the record area after a READ statement.

Indicators Example Programs

This section contains examples of COBOL/400 programs that illustrate the use of
indicators in either a record area or a separate indicator area.

All the programs do the following:

1. Determine the current date.
2. If it is the first day of the month, turn on an option indicator that causes an
   output field to appear and blink.
3. Allow you to press function keys to terminate the program, or turn on response
   indicators and call programs to write daily or monthly reports.

Figure 56 on page 148 shows a program that uses indicators in the record area
but does not use the INDICATORS phrase in any I/O statement. Figure 55 on
page 147 shows the associated DDS for the file.

Figure 57 on page 150 shows a program that uses indicators in the record area
and the INDICATORS phrase in the I/O statements. The associated DDS for
Figure 57 is Figure 55 on page 147.

Figure 59 on page 153 shows a program that uses indicators in a separate indi-
cator area, defined in WORKING-STORAGE by using the Format 2 COPY state-
ment. Figure 58 on page 152 shows the associated DDS for the file.
Figure 60 on page 155 shows a program that uses indicators in a separate indicator area, defined in a table in WORKING-STORAGE. The associated DDS for the file is the same as Figure 58 on page 152.

Figure 55. Example of a Program Using Indicators in the Record Area without Using the INDICATORS Phrase in the I/O Statement–Data Description Specifications

1. The INDARA keyword is not used; indicators are stored in the record area with the data fields.
2. One record format, FORMAT1, is specified.
3. Three indicators are associated with three function keys. Indicator 99 will be set on when you press F3, and so on.
4. One field is defined for input.
5. Indicator 01 is defined to cause the associated constant field to blink if the indicator is on.
6. The function (F) key definitions are documented on the work station display.
Figure 56 (Part 1 of 2). Example of a Program Using Indicators in the Record Area without Using the INDICATORS Phrase in the I/O Statement–COBOL Source Program
The separate indicator area attribute, SI, is not coded in the ASSIGN clause.

The Format 2 COPY statement defines data fields and indicators in the record area.

Because the file does not have a separate indicator area, response and option indicators are defined in the order in which they are used in the DDS, and the indicator numbers are treated as documentation.

All indicators used by the program are defined with meaningful names in data description entries in WORKING-STORAGE. Indicator numbers are omitted here because they have no effect.

For each indicator, a meaningful level-88 condition-name is associated with a value for that indicator.

Initialize group level to zeros.

IN01 in WORKING-STORAGE is set on if it is the first day of the month.

Indicators appropriate to the output of FORMAT1 are copied to the record area.

FORMAT1 is written to the work station display with both data and indicator values in the record area.

The INDICATORS phrase is not necessary because there is no separate indicator area and indicator values have been set in the record area through the previous MOVE CORRESPONDING statement.

FORMAT1, including both data and indicators, is read from the display.

The response indicators for FORMAT1 are copied from the record area to the data description entries in WORKING-STORAGE.

If F5 has been pressed, a program call is processed.
Figure 57 (Part 1 of 2). Example of a Program Using Indicators in the Record Area and the INDICATORS phrase in the I/O Statements–COBOL Source Program
The separate indicator area attribute, SI, is not coded in the ASSIGN clause.

2. The Format 2 COPY statement defines data fields and indicators in the record area.

3. Because the file does not have a separate indicator area, response and option indicators are defined in the order in which they are used in the DDS, and the indicator numbers are treated as documentation.

4. All indicators used by the program are defined with meaningful names in data description entries in WORKING-STORAGE. Indicator numbers are omitted here because they have no effect. Indicators should be defined in the order needed by the display file.

5. IN01 in WORKING-STORAGE is set on if it is the first day of the month.

6. FORMAT1 is written to the workstation display:
   - The INDICATORS phrase causes the contents of the variable OPTION-INDICS to be copied to the beginning of the record area.
   - Data and indicator values are written to the workstation display.

7. FORMAT1, including both data and indicators, is read from the workstation display.

8. The INDICATORS phrase causes bytes to be copied from the beginning of the record area to RESPONSE-INDICS.

9. If F5 has been pressed, a program call is processed.
The INDARA keyword is specified; indicators are stored in a separate indicator area, not in the record area. Except for this specification, the DDS for this file is the same as that shown in Figure 55 on page 147.
Chapter 8. Transaction Files

Figure 59 (Part 1 of 2). COBOL Listing Using Indicators in a Separate Indicator Area

5763CB1 V3ROMS A5/400 COBOL Source

STMT SEQUENCER A.1.B.+.2.+.3.+.4.+.5.+.6.+.7. IDENTCN $ COPYNAME $ CHG $ DATE

1 000100 IDENTIFICATION DIVISION. 03/09/94
2 000200 PROGRAM-ID. XMPLE71. 03/22/94
000300+ SAMPLE PROGRAM - FILE WITH SEPARATE INDICATORS AREA 03/09/94
3 000400 AUTHOR. PROGRAMMER NAME. 03/09/94
4 000500 INSTALLATION. TORONTO COBOL DEVELOPMENT CENTRE. 03/09/94
5 000600 DATE-WRITTEN. 12/08/88. 03/09/94
6 000700 DATE-COMPILED. 05/24/94 12:53:17.
7 000800 ENVIRONMENT DIVISION. 03/09/94
8 000900 CONFIGURATION SECTION. 03/09/94
9 001000 SOURCE-COMPUTER. IBM-AS400. 03/09/94
10 001100 OBJECT-COMPUTER. IBM-AS400. 03/09/94
11 001200 INPUT-OUTPUT SECTION. 03/09/94
12 001300 FILE-CONTROL. 03/09/94
13 001400 SELECT DISPFILE 03/09/94
14 001500 ASSIGN TO WORKSTATION-DSPFILE-SI 03/22/94
15 001600 ORGANIZATION IS TRANSACTION 03/09/94
16 001700 ACCESS IS SEQUENTIAL. 03/09/94
17 001800 DATA DIVISION. 03/09/94
18 002000 FILE SECTION. 03/09/94
19 002100 FD DISPFILE 03/09/94
20 002200 LABEL RECORDS ARE OMITTED 03/09/94
21 002300 DATA RECORD IS DISP-REC. 03/09/94
22 002400 01 DISP-REC. 03/09/94
23 002500 COPY DDS-ALL-FORMATS OF DSPFILE. 03/22/94
24 +000001 05 DSPFILE-RECORD PIC X(5). <-ALL-FMTS
  +000002+ INPUT FORMAT:FORMAT1 FROM FILE DSPFILE OF LIBRARY XMPLIB <-ALL-FMTS
  +000003+ <-ALL-FMTS
25 +000004 05 FORMAT1-I REDORENS DSPFILE-RECORD. <-ALL-FMTS
26 +000005 06 DEPTNO PIC X(5). <-ALL-FMTS
  +000006+ OUTPUT FORMAT:FORMAT1 FROM FILE DSPFILE OF LIBRARY XMPLIB <-ALL-FMTS
  +000007+ <-ALL-FMTS
+000008+ 05 FORMAT1-O REDORENS DSPFILE-RECORD. <-ALL-FMTS
  002600
27 002700 WORKING-STORAGE SECTION.
28 002800 01 CURRENT-DATE.
29 002900 05 CURR-YEAR PIC 9(2).
30 003000 05 CURR-MONTH PIC 9(2).
31 003100 05 CURR-DAY PIC 9(2).
32 003200
33 003300 77 IND-OFF PIC 1 VALUE B"0".
34 003400 77 IND-ON PIC 1 VALUE B"1".
35 003500 01 DSPFILE-INDICS.
36 003600 COPY DDS-ALL-FORMATS-INDIC OF DSPFILE. 003600
36 +000001 05 DSPFILE-RECORD. <-ALL-FMTS
  +000002+ INPUT FORMAT:FORMAT1 FROM FILE DSPFILE OF LIBRARY XMPLIB <-ALL-FMTS
  +000003+ <-ALL-FMTS
37 +000004 06 FORMAT1-INDIC. <-ALL-FMTS
38 +000005 07 IN51 PIC 1 INDIC 51. 000005
39 +000006+ DAILY REPORT <-ALL-FMTS
40 +000007 07 IN52 PIC 1 INDIC 52. <-ALL-FMTS
+000008+ MONTHLY REPORT <-ALL-FMTS
41 +000009 07 IN99 PIC 1 INDIC 99. <-ALL-FMTS
+000010+ END OF PROGRAM <-ALL-FMTS
+000011+ OUTPUT FORMAT:FORMAT1 FROM FILE DSPFILE OF LIBRARY XMPLIB <-ALL-FMTS
+000012+ <-ALL-FMTS
41 +000013 06 FORMAT1-O-INDIC. <-ALL-FMTS
42 +000014 07 INO1 PIC 1 INDIC 01.
  003700
43 003800 PROCEDURE DIVISION.
44 003900
45 004000 MAIN-PROCESS.
  004100
46 004200 OPEN I-O DISPFILE.
45 004300 ACCEPT CURRENT-DATE FROM DATE.
46 004400 MOVE IND-OFF TO IN99 IN FORMAT1-I-INDIC.
47 004500 PERFORM DISPLAY-SCREEN THRU READ-AND-PROCESS-SCREEN
  004600 UNTIL IN99 IN FORMAT1-I-INDIC = IND-ON.
48 004700 CLOSE DISPFILE.
49 004800 STOP RUN.
  004900
50 005000 DISPLAY-SCREEN.
  005100
51 005200 MOVE ZEROS TO FORMAT1-O-INDIC.
51 005300 IF CURR-DAY = 01 THEN
52 005400 MOVE IND-ON TO INO1 IN FORMAT1-O-INDIC. 005400
The separate indicator area attribute, SI, is specified in the ASSIGN clause.

The Format 2 COPY statement generates data descriptions in the record area for data fields only. The data description entries for the indicators are not generated because a separate indicator area has been specified for the file.

The Format 2 COPY statement, with the INDICATOR attribute, INDIC, defines data description entries in WORKING-STORAGE for all indicators used in the DDS for the record format for the file.

Because the file has a separate indicator area, the indicator numbers used in the data description entries are not treated as documentation.

IN01 in the separate indicator area for FORMAT1 is set on if it is the first day of the month.

The INDICATORS phrase is required to send indicator values to the work station display.

The INDICATORS phrase is required to receive indicator values from the work station display. If you have pressed F5, IN51 is set on.

If IN51 has been set on, a program call is processed.
Figure 60 (Part 1 of 2). Example of a Program Using Indicators in a Separate Indicator Area, Defined in a Table in WORKING-STORAGE.
The separate indicator area attribute, SI, is specified in the ASSIGN clause.

2. The Format 2 COPY statement generates fields in the record area for data fields only.

3. A table of 99 Boolean data items is defined in WORKING-STORAGE. The INDICATOR clause for this data description entry causes these data items to be associated with indicators 1 through 99 respectively. The use of such a table may result in improved performance as compared to the use of a group item with multiple subordinate entries for individual indicators.

4. A series of data items is defined in WORKING-STORAGE to provide meaningful subscript names with which to refer to the table of indicators. The use of such data items is not required.

5. INDIC-TABLE (01) in the separate indicator area for FORMAT1 is set on if it is the first day of the month.

6. The INDICATOR phrase is required to send indicator values to the work station display.

7. The INDICATOR phrase is required to receive indicator values from the work station display. If F5 has been pressed, INDIC-TABLE (51) will be set on.

8. If INDIC-TABLE (51) has been set on, program DAILY is called.

Subfiles

Subfiles can be specified in the DDS for a display file to allow you to handle multiple records of the same type on a display. See Figure 61 on page 157 for an example of a subfile display. A subfile is a group of records that are read from or written to a display device. The program processes one record at a time, but the operating system and the work station send and receive blocks of records. If more records are transmitted than can be shown on the display at one time, the work station operator can page through the block of records without returning control to the program.
Records to be included in a subfile are specified in the DDS for the file. The number of records that can be contained in a subfile must also be specified in the DDS. One file can contain more than one subfile; however, only twelve subfiles can be active concurrently for a device. Twelve subfiles can be displayed on a device at the same time.

The DDS for a subfile consists of two record formats: a subfile record format and a subfile control record format.

The subfile record format contains the field descriptions for the records in the subfile. Specifications of the subfile record format on a READ, WRITE, or REWRITE causes the specified subfile record to be processed, but does not directly affect the displayed data.

Specification of the subfile control record format on the READ or WRITE statement causes the physical read, write, or setup operations of a subfile to take place. Figure 62 on page 159 shows an example of the DDS for a subfile record format, and Figure 63 on page 161 shows an example of the DDS for a subfile control record format.

For a description of how the records in a subfile can be displayed and for a description of the keywords that can be specified for a subfile, see the Data Management Guide and also the DDS Reference.

---

### Figure 61. Subfile Display

To use a subfile for a display file in a COBOL program, you must specify the SUBFILE phrase with the input/output operation. Valid subfile operations are:

- **READ SUBFILE file-name RECORD**
- **WRITE SUBFILE record-name**
- **REWRITE SUBFILE record-name**.
Subfiles can be processed sequentially with the READ SUBFILE NEXT MODIFIED statement, or processed randomly by specifying a relative key value. A relative key is an unsigned number that can be used directly by the system to locate a record in a file.

The TRANSACTION file must be an externally defined file. In COBOL, all access to the subfile is done with a relative record number. If the SUBFILE phrases are used with a TRANSACTION file, the SELECT statement in the Environment Division must state that ACCESS MODE IS DYNAMIC and must specify the RELATIVE KEY to be used.

If more than one display device is acquired by a display file, there is a separate subfile for each individual display device. If a subfile has been created for a particular display device acquired by a TRANSACTION file, all input operations that refer to a record format for the subfile are performed against the subfile belonging to that device. See the discussion on the TERMINAL phrase on page 182 of this chapter for information about how to determine which device is used. Any operations that reference a record format name that is not designated as a subfile are processed as an input/output operation directly to the display device.

### Use of Subfiles

Some typical uses of subfiles include:

<table>
<thead>
<tr>
<th>Use</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display Only</td>
<td>The work station user reviews the display.</td>
</tr>
<tr>
<td>Display With Selection</td>
<td>The user requests more information about one of the items on display.</td>
</tr>
<tr>
<td>Modification</td>
<td>The user modifies one or more of the records.</td>
</tr>
<tr>
<td>Input Only (with no validity checking)</td>
<td>A subfile is used for a data-entry function.</td>
</tr>
<tr>
<td>Input Only (with validity checking)</td>
<td>A subfile is used for a data-entry function, and the records are checked as well.</td>
</tr>
<tr>
<td>Combination of Tasks</td>
<td>A subfile can be used as a display with modification.</td>
</tr>
</tbody>
</table>
Figure 62. Data Description Specifications for a Subfile Record Format

The data description specifications (DDS) for a subfile record format describe the records in the subfile:

1. The SFL keyword identifies the record format as a subfile.
2. The line and position entries define the location of the fields on the display.
3. The VALUES keyword specifies that the user can only specify *YES or *NO as values for the ACPPMT field.
4. The usage entries define whether the named field is to be an output (O), input (I), output/input (B), or hidden (H) field.
5. The entry CHECK(FE) specifies that the user cannot skip to the next input field without pressing one of the field exit keys.
The entry AUTO(RAB) specifies that data entered into the field AMPAID is to be automatically right-justified, and the leading characters are to be filled with blanks.

The entry CMP(GT 0) specifies that the data entered for the field AMPAID is to be compared to zero to ensure that the value is greater than zero.

The EDTCDE keyword specifies the desired editing for output field OVRPMT. EDTCDE(1) indicates that the field OVRPMT is to be printed with commas, decimal point, and no sign. Also, a zero balance will be printed, and leading zeros will be suppressed.

The DSPATR keyword is used to specify the display attributes for the named field when the corresponding indicator status is true. The attributes specified are:

- BL (blink)
- RI (reverse image)
- PR (protect)
- MDT (set modified data tag)
- ND (nondisplay).
Figure 63. Data Description Specifications for a Subfile Control Record Format

The subfile control record format defines the attributes of the subfile, the search input field, constants, and command keys. The keywords used indicate the following:

1. SFLCTL identifies this record as a subfile control record and names the associated subfile record (SUBFILE1).
2. SFLSIZ indicates the total number of records to be included in the subfile (17).
3. SFLPAG indicates the total number of records in a page (17).
4. SFLCLR indicates when the subfile should be cleared (when indicator 61 is on).
5. SFLDSP indicates when to display the subfile (when indicator 62 is on).
6. SFLDSPCTL indicates when to display the subfile control record (when indicator 62 is on).
The LOCK keyword prevents the work station user from using the keyboard when the CONTROL1 record format is initially displayed.

HELP allows the user to press the Help key and sets indicator 99 on.

SFLMSG identifies the constant as a message that is displayed if indicator 99 is on.

In addition to the control information, the subfile control record format defines the constants to be used as column headings for the subfile record format. Refer to Figure 63 on page 161 for an example of the subfile control record format.

Multiple Device Files and Single Device Files

A multiple device file is either a display file or an intersystem communications function (ICF) file. A multiple device file can acquire more than one program device. For an example of the use of multiple device files, see Figure 64 on page 163.

A single device file is a device file created with only one program device defined for it. Printer files, diskette files and tape files are single device files. Display files and intersystem communication function (ICF) files created with a maximum number of one program device are also single device files.

A display file can have multiple program devices when the MAXDEV parameter of the CRTDSPF command is greater than 1. If you specify *NONE for the DEV parameter of this command, you must supply the name of a display device before you use any fields that are related to the file.

For more information about how to create and use a display file, see the Data Management Guide.

ICF files can have multiple program devices when the MAXPGMDEV parameter of the CRTICFF command is greater than 1. For more information about how to create and use ICF files, see the ICF Programmer’s Guide.

COBOL determines at run time whether a file is a single device file or a multiple device file, based on whether the file is capable of having multiple devices. The actual number of devices acquired does not affect whether a file is considered a single or multiple device file. Whether a file is a single or a multiple device file is not determined at compilation time; this determination is based on the current description of the display or ICF file.

For multiple device files, if a particular program device is to be used in an I/O statement, that device is specified by the TERMINAL phrase. The TERMINAL phrase can also be specified for a single device file.

The following pages contain an example illustrating the use of multiple device files. The program uses a display file, and is intended to be run in batch mode. The program acquires terminals and invites those terminals using a sign-on display. After the terminals are invited, they are polled. If nobody signs on before the wait time expires, the program ends. If you enter a valid password, you are allowed to update an employee file by calling another COBOL program. Once the update is complete, the device is invited again and the terminals are polled again.
The format SIGNON has the keyword INVITE associated with it. This means that, if format SIGNON is used in a WRITE statement, the device to which it is writing will be invited.
### Figure 64 (Part 2 of 3). Example of the Use of Multiple Device Files

```
A
A* DD$ FOR THE PHYSICAL FILE PASSWORD
A*                    UNIQUE
A* R  PASSWORDS
A*  PASSKEY  16
A*  PASSWORD  16
A* K  PASSKEY
```

**International Business Machines**

**AS/400 DATA DESCRIPTION SPECIFICATIONS**

**COBOL/400 User's Guide**
### Figure 64 (Part 3 of 3). Example of the Use of Multiple Device Files

<table>
<thead>
<tr>
<th>Description</th>
<th>Page of</th>
</tr>
</thead>
<tbody>
<tr>
<td>A*</td>
<td></td>
</tr>
</tbody>
</table>

**A**

**DLS FOR THE PHYSICAL FILE TERM**

**A** WHICH CONTAINS THE LIST OF TERMINALS

**A**

**TERM**

**TERM** 10

---

**Chapter 8. Transaction Files 165**
Figure 65 (Part 1 of 4). COBOL Source Listing for Multiple Device File Support
Figure 65 (Part 2 of 4). COBOL Source Listing for Multiple Device File Support
PROCEDURE DIVISION.

001280
001290 DECLARATIVES.
001300
001310 MULTIPLE-SECTION SECTION.
001320 USE AFTER STANDARD EXCEPTION PROCEDURE ON MULTIPLE-FILE.
001330
001340 MULTIPLE-PARAGRAPH.

101 001350 WRITE  PRINTER-REC FROM HEADER-LINE AFTER ADVANCING PAGE.
102 001360 MOVE "FILE NAME IS:" TO DESCRIPTION OF DETAIL-LINE.
103 001370 MOVE "MULTIPLE FILE" TO DETAIL-VALUE OF DETAIL-LINE.
104 001380 WRITE  PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 5 LINES.
105 001390 MOVE "FILE STATUS IS:" TO DESCRIPTION OF DETAIL-LINE.
106 001400 MOVE MULTIPLE-FS1 TO DETAIL-VALUE OF DETAIL-LINE.
107 001410 MOVE "FILE NAME IS:" TO DESCRIPTION OF DETAIL-LINE.
108 001420 MOVE "FILE STATUS IS:" TO DETAIL-LINE.
109 001430 MOVE MULTIPLE-FS2 TO DETAIL-VALUE OF DETAIL-LINE.
110 001440 WRITE  PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 2 LINES.
111 001450 STOP RUN.

001500
001510 TERMINAL-SECTION SECTION.
001520 USE AFTER STANDARD EXCEPTION PROCEDURE ON TERMINAL-FILE.
001530 TERMINAL-PARAGRAPH.

116 001540 WRITE  PRINTER-REC FROM HEADER-LINE AFTER ADVANCING PAGE.
117 001550 MOVE "FILE NAME IS:" TO DESCRIPTION OF DETAIL-LINE.
118 001560 MOVE "TERMINAL FILE" TO DETAIL-VALUE OF DETAIL-LINE.
119 001570 WRITE  PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 5 LINES.
120 001580 MOVE "FILE STATUS IS:" TO DESCRIPTION OF DETAIL-LINE.
121 001590 MOVE TERMINAL-FS1 TO DETAIL-VALUE OF DETAIL-LINE.
122 001600 WRITE  PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 2 LINES.
123 001610 STOP RUN.

001620
001630 PASSWORD-SECTION SECTION.
001640 USE AFTER STANDARD EXCEPTION PROCEDURE ON PASSWORD-FILE.
001650 PASSWORD-PARAGRAPH.

124 001660 WRITE  PRINTER-REC FROM HEADER-LINE AFTER ADVANCING PAGE.
125 001670 MOVE "FILE NAME IS:" TO DESCRIPTION OF DETAIL-LINE.
126 001680 MOVE "PASSWORD FILE" TO DETAIL-VALUE OF DETAIL-LINE.
127 001690 WRITE  PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 5 LINES.
128 001700 MOVE "FILE STATUS IS:" TO DESCRIPTION OF DETAIL-LINE.
129 001710 MOVE PASSWORD-FS1 TO DETAIL-VALUE OF DETAIL-LINE.
130 001720 WRITE  PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 2 LINES.
131 001730 STOP RUN.

001740
001750 END DECLARATIVES.
001760
001770******************************************************************************************
001780* MAIN PROGRAM LOGIC BEGINS HERE *
001790******************************************************************************************
001800
001810 MAIN-LINE SECTION.
001820 MAIN-LINE-PARAGRAPH.

132 001830 OPEN I-O MULTIPLE-FILE 20
001840 INPUT TERMINAL-FILE 1-0 PASSWORD-FILE
001850 OUTPUT PRINTER-FILE.
001870
133 001880 MOVE 1 TO COUNTER.
134 001890 SET NOT-END-OF-TERMINAL-LIST TO TRUE.
001900 PERFORM FILL-TERMINAL-LIST UNTIL END-OF-TERMINAL-LIST.

135 001910 PERFORM ACQUIRE-AND-INVITE-TERMINALS.
136 001920 VARYING COUNTER FROM 1 BY 1
137 001930 MOVE 1 TO COUNTER.
138 001940 SET DATA-AVAILABLE TO TRUE.
001950 PERFORM PULL-TERMINALS UNTIL NO-DATA-AVAILABLE.
001960 PERFORM DROP-TERMINALS.
139 001970 VARYING COUNTER FROM 1 BY 1
140 001980 UNTIL COUNTER GREATER THAN NO-OF-TERMINALS.

Figure 65 (Part 3 of 4). COBOL Source Listing for Multiple Device File Support
Device File Attributes

1. ATTR is the mnemonic-name associated with the function-name ATTRIBUTE-DATA. ATTR is used in the ACCEPT statement to obtain attribute data for the TRANSACTION file MULTIPLE-FILE. See item 9A.

2. File MULT must have been created using the CRTDSPF command, where the DEV parameter has a value of *NONE and the MAXDEV parameter has a value greater than 1. The WAITRCD parameter specifies the wait time for READ operations on the file. The WAITRCD parameter must have a value greater than 0.

3. MULTIPLE-FS2 is the extended file status for the TRANSACTION file MULTIPLE-FILE. This variable has been declared in the WORKING- STORAGE section of the program. See item 7.

4. MULTIPLE-CONTROL-AREA is the control area for the TRANSACTION file MULTIPLE-FILE. This variable is used to determine which program device was used to sign on. See item 15.

5. The data description for MULTIPLE-REC has been defined using the COPY DDS statement.

Note: Only the fields that are copied are named fields. Refer to the DDS of this example for comments regarding the DDS used.
Format SIGNON is the format with the INVITE keyword. This is the format that will be used to invite devices via the WRITE statement.

This is the declaration for the extended file-status MULTIPLE-FS2. It is a 4-byte field that is subdivided into a major return code (first 2 bytes) and a minor return code (last 2 bytes).

STATION-ATTR is where the ACCEPT statement contains the attribute data for the TRANSACTION file MULTIPLE-FILE. See item 9A.

In this statement, the extended file status MULTIPLE-FS2 is being written.

This is an example of accepting attribute-data for the TRANSACTION file MULTIPLE-FILE. Because there is no interest in a specific program device, but rather the last program device used, the FOR phrases are not used with the ACCEPT.

This statement opens the TRANSACTION file MULTIPLE-FILE. Because the ACQPGMDEV parameter of the CRTDSPF command has the value "NONE, no program devices are implicitly acquired when this file is opened.

This statement acquires the program device contained in the variable LIST-OF-TERMINALS (COUNTER), for the TRANSACTION file MULTIPLE-FILE.

This WRITE statement is inviting the program device specified in the TERMINAL phrase. The format SIGNON has the DDS keyword INVITE associated with it. Refer to item 12.

This READ statement will read from any invited program device. See item 12. If the wait time expires before anyone inputs to the invited devices, the extended file status will be set to "0310" and processing will continue. See item 14.

In this statement, the extended file status for MULTIPLE-FILE is being checked to see if the wait time expired.

The program device name stored in the control area is used to determine which program device was used to sign on. See item 14.

This DROP statement detaches the program device contained in the variable LIST-OF-TERMINALS from the TRANSACTION file MULTIPLE-FILE.
Environment Division

File-Control Entry

The TRANSACTION file must be named by a file-control entry in the FILE-CONTROL paragraph. This entry also specifies other information related to the file.

Format
ASSIGN Clause
The ASSIGN clause associates the TRANSACTION file with a display file or ICF file through the use of assignment-name-1.

Assignment-name-1 has the following structure:

```
<table>
<thead>
<tr>
<th>ASSIGN</th>
<th>WORKSTATION</th>
<th>-file-name</th>
<th>-SI</th>
</tr>
</thead>
<tbody>
<tr>
<td>TO</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Device specifies the type of device associated with the file. The value must be WORKSTATION.

The AS/400 file name is a one-to-ten character external name of the display file or ICF file specified on the create device file commands, CRTDSPF or CRTICFF.

The attribute -SI is used to specify the file level option for a separate indicator area. See “Using Indicators with Transaction Files” on page 142 for further details.

The second and subsequent assignment-names are syntax-checked, but are treated as documentation.

ORGANIZATION Clause
The ORGANIZATION clause specifies the logical structure of a file. TRANSACTION organization signifies interaction between the program and either a workstation user or another system.

**TRANSACTION Organization:** TRANSACTION processing is defined as the random arrival of a record from one of multiple possible sources followed by appropriate processing, and finally, by the output of results or feedback information of some type to the source of the record.

In some cases, all records are homogeneous; that is, a logical transaction is completed with one exchange of records. In other situations, a series of records is passed back and forth in a logical progression with various record types either being selected by the initiator or as part of the processing based on input data values.

Each transaction can be processed by a different program, or multiple transactions can be processed by the same program, depending on the system environment.

The initiation of a transaction can cause a program to be scheduled to process the transaction.

A transaction can consist of a series of alternating requests and responses (a dialogue). Each request and response can consist of multiple logical records.
ACCESS MODE Clause

For files with TRANSACTION organization, the access mode can be SEQUENTIAL or DYNAMIC.

Note: Dynamic processing is a method of reading from or writing to a file in a nonsequential order and reading from a file in a sequential order with the same OPEN statement.

When ACCESS IS SEQUENTIAL is specified or implied, the format name contained in the format name field of the control area specifies which record was accessed. When ACCESS IS SEQUENTIAL is specified for a TRANSACTION file, do not specify the RELATIVE KEY data item.

When ACCESS IS DYNAMIC is specified, records in the file can be accessed sequentially or randomly, depending on the form of the specific input/output request. Random accessing of a TRANSACTION file is only valid if subfile processing is being performed. For subfile processing, you must specify ACCESS IS DYNAMIC.

RELATIVE KEY Clause

The RELATIVE KEY clause specifies the relative record number for a specific record in a subfile. The RELATIVE KEY data item, data-name-3, must be defined as an unsigned integer and cannot be scaled. Also, the data item must not be defined in a record description entry associated with the TRANSACTION file.

FILE STATUS Clause

Data-name-5 identifies the extended-file-status data item, which contains major and minor return codes. These major and minor return codes can, in some cases, indicate I/O errors when the file status code does not. After an I/O operation is performed on an unopened file, the extended file status will have a value of zeros.

For more information about the FILE STATUS clause, refer to “File Status and Feedback Areas” on page 103. General considerations about the FILE STATUS clause and data-name-1 are described in Part 2 of the COBOL/400 Reference in the section, “FILE STATUS Clause.”

For information about the role of file status in error handling, refer to Chapter 6, “COBOL/400 Exception and Error Handling” on page 69.

Data-name-5 must be defined in the Data Division as a 4-byte alphanumeric data item, and must not be defined in the File Section. The first 2 bytes of the extended-file-status data item contain the major return code, and the second 2 bytes contain the minor return code. Return codes are moved into data-name-5 after any input or output operation (except the ACCEPT or CLOSE statement) on the TRANSACTION file. The values placed in data-name-5 can also be accessed by the ACCEPT statement using the I-O-FEEDBACK function-name. For more information about the major and minor return codes, see the Data Management Guide and the ICF Programmer’s Guide.
CONTROL-AREA Clause

The CONTROL-AREA clause specifies device-dependent and system-dependent information that is used to control input/output operations for TRANSACTION files.

Data-name-6 is a CONTROL-AREA data item that must be defined in the LINKAGE SECTION or WORKING-STORAGE SECTION. Data-name-6 is assumed to have the following format:

01 data-name-6.
   02 function-key PIC X(2).
       (Function key feedback field)
   02 device-name PIC X(10).
       (Program device name)
   02 record-format PIC X(10).
       (Record format)

Data-name-6 must be 2, 12, or 22 characters long. Based upon the length of data-name-6, the compiler assumes the availability of key feedback bytes, the program device name, and record format.

Programming Note: For an ICF file, the actual name of a device may be different from the program device name (data-name-11).

Information is moved into data-name-6 for each READ operation from a file that has been assigned to a WORKSTATION device type. The information is valid only if the READ operation is successfully completed (provided the wait time has not expired). The information is in the fixed format as shown in the following example:

FILE-CONTROL.
SELECT SCREEN-FILE
   ASSIGN TO WORKSTATION-MYFMTS
   ORGANIZATION IS TRANSACTION
   CONTROL-AREA IS
       TRANSACTION-CONTROL-AREA.
;
WORKING-STORAGE SECTION.
01 TRANSACTION-CONTROL-AREA.
   * FEEDBACK ITEM
      02 FUNCTION-KEY PIC XX.
      02 TERMINAL-ID PIC X(10).
      02 FORMAT-NAME PIC X(10).
Each field in the TRANSACTION-CONTROL-AREA data item in the example is described as follows:

- **FUNCTION-KEY**: A two-digit number inserted in the field by the work station interface that identifies the function key the operator pressed to initiate the transaction. The codes are as follows:

  00  Enter key
  01-24 Function keys 1 through 24
  90  Roll Up/Page Down key
  91  Roll Down/Page Up key
  92  Print key
  93  Help key
  94  Clear key
  95  Home key
  99  Undefined

  Any function keys for which feedback information is desired must be defined for the display file using DDS.

- **TERMINAL-ID**: The program device name.

- **FORMAT-NAME**: The DDS record format name that was referenced by the last I/O statement run.

---

**Data Division**

**File Description Entry**

A file description entry consists of a level indicator (FD), a file name, and a series of independent clauses. For a TRANSACTION file, the independent clauses allowed are the RECORD CONTAINS clause, the LABEL RECORDS clause, and the DATA RECORDS clause.
The LABEL RECORDS clause specifies whether or not labels are present. This clause is required in every file description entry. This clause is syntax-checked, but is treated as documentation.

**Boolean Data Items**

The use of Boolean data and the use of indicators are described under “Data Description Entry–Boolean Data” on page 144.

**Procedure Division**

**Procedure Division Concepts**

The COBOL/400 language provides a number of extensions to PROCEDURE DIVISION statements to support TRANSACTION processing. The sections that follow describe the statements involved and their usage.
The ACCEPT statement retrieves information (attribute data) about a particular program device associated with a TRANSACTION file.

This format of the ACCEPT statement can only be used for files with an organization of TRANSACTION. Mnemonic-name must be associated with the function-name ATTRIBUTE-DATA in the SPECIAL-NAMES paragraph.

If file-name is not specified, the default file for the ACCEPT statement is the first TRANSACTION file specified in a SELECT clause of the FILE-CONTROL paragraph.

Literal-1 or the contents of identifier-2, if specified, indicates the program device name for which attribute data is made available. This device must be defined by a CRTDSPF, ADDICFDEVE, or OVRICFDEVE CL command. The device does not actually have to be acquired. Literal-1, if specified, must be nonnumeric and 10 characters or fewer in length. The contents of identifier-2, if specified, must be an alphanumeric data item 10 characters or fewer in length. If an incorrect program device name is specified, or if the file is not open at the time the ACCEPT statement is processed, message LBE7205 ACCEPT ATTRIBUTE-DATA statement has failed (C D F) is issued and processing terminates.

If both FOR phrases are omitted (indicating the default TRANSACTION file is being used), the ACCEPT statement uses the program device from which a READ, WRITE, REWRITE, or ACCEPT (Attribute Data) operation on the default file was most recently performed. If the only prior operation on the file was an OPEN, the ACCEPT statement uses the program device implicitly acquired by the file when the file was opened. When both FOR phrases are omitted, a program device must have been acquired to use this particular format of the ACCEPT statement.

Program device attributes are moved into identifier-1 from the appropriate attribute data format, according to the rules for a group MOVE without the CORRESPONDING phrase.
You can make use of multiple display files along with ordinary files in a program that includes an Extended ACCEPT or Extended DISPLAY statement. (See the COBOL/400 Reference for more information.)

Attribute Data Formats
The attribute data retrieved by the ACCEPT statement has two different formats, depending if the data is for a work station or for a communications device.

The ATTRIBUTE-DATA mnemonic name can be used only to obtain information about a program device for a TRANSACTION file. Attribute data does not provide information about the status of a completed or attempted I/O operation. To obtain information about I/O operations, use the Format 3 ACCEPT statement with the I-O-FEEDBACK or OPEN-FEEDBACK mnemonic names. For more information about these mnemonic names, see the “SPECIAL NAMES Paragraph” section of the COBOL/400 Reference.

ACQUIRE Statement
The ACQUIRE statement acquires a program device for a TRANSACTION file.

ACQUIRE Statement – TRANSACTION File

Literal or the contents of identifier indicates the program device name to be acquired by the specified file. Literal, if specified, must be nonnumeric and 10 characters or fewer in length. Identifier, if specified, must refer to an alphanumeric data item 10 characters or fewer in length.

File-name must be the name of a file with an organization of TRANSACTION, and the file must be open when the ACQUIRE statement is run. A compilation error message is issued if the organization is not TRANSACTION.

For a description of conditions that must be met before a communications device can be acquired, see the ICF Programmer’s Guide. For more information about the requirements for displays, see the Data Management Guide.

Successful completion of the ACQUIRE operation makes the program device available for input and output operations.

If the ACQUIRE operation is unsuccessful, the file status value is set to 9H and the USE AFTER EXCEPTION/ERROR procedure is called (if specified). For more information, refer to Chapter 6, “COBOL/400 Exception and Error Handling.”

Only one program device can be implicitly acquired when a file is opened. If a file is an ICF file, the single implicitly acquired program device is determined by the ACQPGMDEV parameter of the CRTICFF command. If the file is a display file, the
single implicitly acquired program device is determined by the first entry in the DEV parameter of the CRTDSPF command. Additional program devices must be explicitly acquired.

A program device is explicitly acquired by using the ACQUIRE statement. For an ICF file, that device must have been defined to the file with the ADDICFDEVE or OVRICFDEVE CL command before the file was opened. For display files there is no such requirement. That is, the device named in the ACQUIRE statement does not have to be specified in the DEV parameter of the CRTDSPF command, the CHGDSPF command, or the OVRDSPF command. For a display file, the program device name must match the display device.

The ACQUIRE statement can also be used as an aid in recovering from I/O errors. For more information, see the “ACQUIRE Statement” section of the COBOL/400 Reference.

For more information about these commands, see the CL Reference.

CLOSE Statement
The CLOSE statement terminates the processing of volumes and files, with optional lock where applicable.

CLOSE Statement

Drop Statement
The DROP statement releases a program device that has been acquired by a TRANSACTION file.
Literal or the contents of identifier indicates the program device name of the device to be dropped. Literal, if specified, must be nonnumeric and 10 characters or fewer in length. Identifier, if specified, must refer to an alphanumeric data item, 10 characters or fewer in length.

File-name must refer to a file with an organization of TRANSACTION, and the file must be open to be used in the DROP statement. If no DROP statement is issued, program devices attached to a TRANSACTION file are implicitly released when that file is finally closed.

Program devices specified in a DROP statement must have been acquired by the TRANSACTION file, either through an explicit ACQUIRE or through an implicit ACQUIRE at OPEN time.

After successful running of the DROP statement, the program device is no longer available for input or output operations through the TRANSACTION file. The device can be reacquired if necessary. The contents of the record area associated with a released program device are no longer available, even if the device is reacquired.

If the DROP operation is unsuccessful, the USE AFTER EXCEPTION/ERROR procedure is processed (if specified). For more information, refer to Chapter 6, “COBOL/400 Exception and Error Handling.”

The DROP statement can also be used as an aid in recovering from I/O errors. For more information, see the “DROP Statement” section of the COBOL/400 Reference.

OPEN Statement
The OPEN statement initiates the processing of files.

A TRANSACTION file must be opened in the I/O mode. For a further discussion of the OPEN statement, see the COBOL/400 Reference.

The OPEN statement can cause a program device to be implicitly acquired for a TRANSACTION file. For a further discussion about the acquiring of program devices, see the “ACQUIRE Statement” on page 178.
Common Processing Facilities

The following discussion on FORMAT, INDICATORS, SUBFILE, and TERMINAL phrases relates to the READ, REWRITE, and WRITE statements.

**FORMAT Phrase**

The literal or identifier specified must be a character string of 10 characters or fewer in length.

Multiple data records, each with a different format, can be concurrently active for a TRANSACTION file. If the FORMAT phrase is specified, it must specify a valid format name that is defined to the system, and the I/O operation must be performed on a data record of the same format. If the format is an invalid name or if it does not exist, the FILE STATUS data item, if specified, is set to a value of 9K and the contents of the record area are undefined.

**DB-FORMAT-NAME Special Register:** After the running of an input/output statement for a TRANSACTION file, the DB-FORMAT-NAME special register is modified according to the following rules:

- If the input/output operation is successful, the record format name is implicitly moved to the special register after completion of the input/output operation.
- If the input/output operation is unsuccessful, DB-FORMAT-NAME contains the record format name used in the last successful input/output operation.

When the FORMAT phrase is not specified, DB-FORMAT-NAME can be used if the file contains a default record format name. The default value is always moved to the DB-FORMAT-NAME special register.

DB-FORMAT-NAME is implicitly defined as PICTURE X(10).

**INDICATORS Phrase**

The identifier specified in the INDICATORS phrase must be either an elementary Boolean data item specified without the OCCURS clause or a group item that has elementary Boolean data items subordinate to it.

When a data record is written or rewritten, indicators can be written or rewritten with it. The indicators can control how the record is displayed and the various data management functions.

When a data record is read, indicators can be read with it. The indicators can be used to pass information about the data record and how it was entered into your program.

By defining a format using DDS, you determine what functions are to be controlled by indicators, and which indicators control a particular function.

For detailed information on the INDICATORS phrase, refer to “Using Indicators with Transaction Files” on page 142.
**SUBFILE Phrase**

When the SUBFILE phrase is specified, it indicates that all formats referenced by the statement are subfiles. When SUBFILE is not specified in a TRANSACTION I/O statement, it indicates that none of the formats referenced by the statement are subfiles. This information is not verified at compilation time. If it is specified incorrectly, the subfile is processed as a series of input/output operations directly to the display device. When the specified format name exists as a display file format, the READ/WRITE operations complete successfully.

When SUBFILE is not specified, the RELATIVE KEY data item associated with the file, if specified, is not referenced or changed by the I/O operation.

When SUBFILE is specified, a RELATIVE KEY data item must be defined for the file. Its value is referenced, and sometimes changed, by the I/O operation. See each of the statements associated with SUBFILE operations for a detailed description of when and how the RELATIVE KEY data item is changed.

The SUBFILE phrase can be specified only for display files.

**TERMINAL Phrase**

When the TERMINAL phrase is specified, it indicates a specific program device is to be used for a READ, WRITE, or REWRITE operation on a TRANSACTION file.

The TERMINAL phrase can be omitted for I/O operations on single device files, because that device is always used.

If the TERMINAL phrase is omitted for an I/O operation on a TRANSACTION file that has acquired multiple program devices, the program device that last attempted a READ, WRITE, REWRITE, ACQUIRE, DROP, or ACCEPT (Attribute Data) operation on the file is used. If the only prior operation on the file was an OPEN, the default program device used is the program device implicitly acquired by the TRANSACTION file when the file was opened. A run-time error message occurs if no program device has been acquired when the file is opened.

For a READ statement with both the TERMINAL phrase and the NO DATA phrase specified, the imperative-statement in the NO DATA phrase is run only if data is not immediately available from the program device specified by the TERMINAL phrase.

If the TERMINAL phrase is specified and the data-item or literal has a value of blanks, the phrase is treated at run time as if it were not specified.

**READ Statement**

The READ statement makes available a record from a device, using a named format. If the format is a subfile, the READ statement makes available a specified record from that subfile.
Format 4 is used only to read a format that is not a subfile. The RELATIVE KEY data item, if specified in the FILE-CONTROL entry, is not used. The Format 4 READ statement is not valid for a subfile record. However, a Format 4 READ statement for the subfile control record format must be used to place those subfile records that were updated on a display into the subfile.

If the requested data is available, it is returned in the record area. The names of the record format and the program device are returned in the I-O-FEEDBACK area in the CONTROL-AREA.

The READ statement is valid only when there are acquired devices for the file. If a READ is processed and there are no acquired devices, the file status is set to 92 (logic error).
The manner in which the Format 4 READ statement functions depends on:

- If the READ is for a single device file or a multiple device file
- If a specific program device has been requested through the TERMINAL phrase
- If a specific record format has been requested through the FORMAT phrase
- If the NO DATA phrase has been specified.

In the following sections, references to data available or returned include the situation where only the response indicators are set. This also applies even when a separate indicator area is used and the indicators are not returned in the record area for the file.

The following chart shows the possible combinations of phrases and the function performed for a single device file or a multiple device file. For example, if TERMINAL is N, FORMAT is N, and NO DATA is N, the single device is D and multiple device is A.

<table>
<thead>
<tr>
<th>Function</th>
<th>Phrase</th>
<th>Y=Yes</th>
<th>N=No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checked at Compilation</td>
<td>TERMINAL²</td>
<td>N N N N Y Y Y Y</td>
<td>N N Y Y N N Y Y</td>
</tr>
<tr>
<td></td>
<td>FORMAT²</td>
<td>N N Y Y N N Y Y</td>
<td>N Y N Y N Y Y Y</td>
</tr>
<tr>
<td></td>
<td>NO DATA</td>
<td>N Y N Y N Y Y Y</td>
<td>N Y N Y Y Y Y Y</td>
</tr>
<tr>
<td>Determined at Run Time</td>
<td>Single Device</td>
<td>D C D B D C D B</td>
<td>A A D B D C D B</td>
</tr>
<tr>
<td></td>
<td>Multiple Device</td>
<td>D C D B D C D B</td>
<td>A A D B D C D B</td>
</tr>
</tbody>
</table>

Codes A through D are explained below:

**Code A—Read From Invited Program Device (Multiple Device Files only)**

This type of READ receives data from the first invited program device that has data available. Invited program devices are work stations or other communication devices that are invited to send input. The inviting is done by writing to the program device with a format specifying the DDS keyword INVITE. Once an invited program device is actually read from, it is no longer invited. That program device will not be used for input by another READ statement unless reinvited, or unless a READ is directed to it specifying the TERMINAL phrase or FORMAT phrase.

The record format returned from the program device is determined by the system. See the chapter on display device support in the *Data Management Guide* for information on how record format is determined for work stations. See the *ICF Programmer’s Guide* for information on the FMTSLT parameter on the ADDICFDEVE and OVRICFDEVE commands.

This READ can be completed without returning any data in the following cases:

- If there are no invited devices.
- If a controlled cancelation of the job occurs. This results in a file status value of 9A and a major/minor return code value of 0309.

---

² If the phrase is specified and the data item or literal is blank, the phrase is treated at run time as if it were not specified.
• If the NO DATA phrase is omitted, and the specified wait time expires. This results in a file status value of 00 and a major/minor return code value of 0310.

• If the specified wait time is the value entered on the WAITRCD parameter for the file.

• If the NO DATA phrase is specified, and no data is immediately available when the READ is processed.

If data is available, it is returned in the record area. The record format is returned in the I-O-FEEDBACK area and in the CONTROL-AREA. For more information about “Reading from Invited Program Devices,” see the ICF Programmer's Guide.

Code B–Read From One Program Device (Combination not Allowed)

A compilation-time message is issued, and the NO DATA phrase is ignored. See the table entry for the same combination of phrases with the NO DATA phrase omitted.

Code C–Read From One Program Device (with NO DATA phrase)

This function of the READ statement never causes program processing to stop and wait until data is available. Either the data is immediately available or the NO DATA imperative-statement is processed.

This READ function can be used to periodically check if data is available from a particular program device (either the default program device or one specified by the TERMINAL phrase). This checking for data is done in the following manner:

1. The program device is determined as follows:
   a. If the TERMINAL phrase was omitted or contains blanks, the default program device is used. The default program device is the one used by the last attempted READ, WRITE, REWRITE, ACQUIRE, or DROP statement. If none of the above I/O operations were previously issued, the default program device is the first program device acquired.
   b. If the TERMINAL phrase was specified, the indicated program device is used.

2. A check is done to determine if data is available and if the program device is invited.

3. If data is available, that data is returned in the record area and the program device is no longer invited. If no data is immediately available, the NO DATA imperative-statement is run and the program device remains invited.

4. If the program device is not invited, the AT END condition exists and the file status is set to 10.

Code D–Read From One Program Device (without NO DATA Phrase)

This READ always waits for data to be made available. Even if the job receives a controlled cancellation, or a WAITRCD time is specified for the file, the program will never regain control from the READ statement. This READ operation is performed in the following manner:
1. The program device is determined as follows:
   a. If the TERMINAL phrase is omitted or contains a blank value, the default program device is used. The default program device is the program device used by the last attempted READ, WRITE, REWRITE, ACQUIRE, DROP or ACCEPT (Attribute Data) statement. If none of these operations has been done, the program device implicitly acquired when the file was opened is used. If there are no acquired devices, the AT END condition exists.
   b. If the TERMINAL phrase is specified, the indicated program device is used.

2. The record format is determined as follows:
   a. If the FORMAT phrase is omitted or contains blanks, the record format returned is determined by the system. For information on how the record format is calculated for work station devices, refer to the Data Management Guide. For information about how the record format is determined for communications, see the section on the FMTSLT parameter on the ADDICFDEVE and OVRICFDEVE commands in the ICF Programmer's Guide.
   b. If the FORMAT phrase is specified, the indicated record format is returned. If the data available does not match the requested record format, a file status of 9G is set.

3. Program processing stops until data becomes available. The data is returned in the record area after the READ statement is run. If the program device was previously invited, it will no longer be invited after this READ statement.

**INTO Phrase**
The INTO phrase can be specified if:
- Only one record description is subordinate to the file description entry,

OR
- All record names associated with file-name and the data item referenced by identifier-1 describe a group item or an elementary alphanumeric item.

**FORMAT Phrase**
Literal-1 or identifier-2 specifies the name of the record format to be read. Literal-1, if specified, must be nonnumeric and 10 characters or fewer in length. Identifier-2, if specified, must refer to an alphanumeric data item, 10 characters or fewer in length. If identifier-2 contains blanks, the READ statement is run as if the FORMAT phrase were omitted.

**NO DATA Phrase**
When the NO DATA phrase is specified, the READ statement determines if data is immediately available. If data is available, the data is returned in the record area. If no data is immediately available, imperative-statement-1 is processed. The NO DATA phrase prevents the READ statement from waiting for data to become available.
TERMINAL Phrase
Literal-2 or identifier-3 specifies the program device name. Literal-2, if specified, must be nonnumeric and 10 characters or fewer in length. Identifier-3, if specified, must refer to an alphanumeric data item, 10 characters or fewer in length. The program device must have been acquired before the READ statement is processed. If identifier-3 contains blanks, the READ statement is processed as if the TERMINAL phrase were omitted. For a single device file, the TERMINAL phrase can be omitted. The program device is assumed to be that single device.

If the TERMINAL phrase is omitted for a READ of a TRANSACTION file that has acquired multiple program devices, the default program device is used. See the discussion of the TERMINAL phrase on page 182, to see how the default program device is determined.

AT END Phrase
Imperative-statement-2 is performed when the AT END condition is detected.

Note: An AT END condition occurs at the following times:

- During a READ statement for a sequentially accessed file when no next logical record exists in the file, or when the number of significant digits in the relative record number is larger than the size of the relative key data item, or when an optional input file is not present.
- During a RETURN statement when no logical record exists for the associated sort or merge file.
- During a SEARCH statement when the search operation ends without satisfying the condition specified in any of the associated WHEN phrases.

NOT AT END Phrase
This phrase allows you to specify procedures to be performed when the READ operation is successful.

END-READ Phrase
The END-READ phrase serves to explicitly delimit the scope of the statement.
Format 5 is used only to read a format that is a subfile record. The AT END phrase can only be used when the NEXT MODIFIED phrase is specified. The INVALID KEY phrase must not be used when the NEXT MODIFIED phrase is specified.
Format 5 cannot be used for communications devices. If the subfile format of the READ statement is used for a communications device, the READ fails and a file status of 90 is set.

Random Access of Subfile Records: The NEXT MODIFIED phrase must not be used to randomly access records in a subfile. The INVALID KEY phrase can only be used for random access of subfile records.

Sequential Access of Subfile Records: The NEXT MODIFIED phrase must be specified to access subfile records sequentially. The AT END phrase can only be specified with the NEXT MODIFIED phrase.

NEXT MODIFIED Phrase
When NEXT MODIFIED is not specified, the data record made available is the record in the subfile with a relative record number that corresponds to the value of the RELATIVE KEY data item.

When the NEXT MODIFIED phrase is not specified, and if the RELATIVE KEY data item contains a value other than the relative record number of a record in the subfile, the INVALID KEY condition exists and the running of the READ statement is unsuccessful.

When the NEXT MODIFIED phrase is specified, the record made available is the next modified record following the current pointer position in the file. For information about turning on the Modified Data Tag, see the Data Management Guide.

The search for the next modified record begins:

- At the beginning of the subfile if:
  - An I/O operation has been performed for the subfile control record.
  - The I/O operation cleared, initialized, or displayed the subfile.
- For all other cases, with the record following the record that was read by a previous read operation.

The value of the RELATIVE KEY data item is updated to reflect the relative record number of the record made available to the program.

If NEXT MODIFIED is specified and there are no further user-modified records in the subfile, the AT END condition exists. Imperative-statement-2, or an applicable USE AFTER ERROR/EXCEPTION procedure, if any, is then run.

FORMAT Phrase
When a format-name is not specified, the format used is the last record format written to the display device that contains input fields, input/output fields, or hidden fields. If no such format exists for the display file, the format used is the record format of the last WRITE operation to the display device.

Note: An input field is a field specified in a display file or database file that is reserved for information supplied by a user.

If the FORMAT phrase is specified, literal-1 or the contents of identifier-2 must specify a format, which is active for the appropriate program device. The READ statement reads a data record of the specified format.
To ensure correct results, always specify the FORMAT phrase for multiple format files. For more information on the FORMAT phrase, see the Procedure Division, “Common Processing Facilities” on page 181.

TERMINAL Phrase
See Format 4 of the READ Statement for general considerations concerning the TERMINAL phrase.

For a Format 5 READ, if the TERMINAL phrase is omitted for a file that has multiple devices acquired for it, a record is read from the subfile associated with the default program device. See the discussion of the TERMINAL phrase on page 182, to see how the default program device is determined.

INVALID KEY Phrase
If the RELATIVE KEY data item at the time of running the statement contains a value that does not correspond to a relative record number for the subfile, the INVALID KEY condition exists and the running of the statement is unsuccessful. To see what happens next, refer to the diagrams on pages 76 through 78.

For a Format 5 READ, you should specify the INVALID KEY phrase if the NEXT MODIFIED phrase is not specified and there is no applicable USE procedure specified for the file name.

NOT INVALID KEY Phrase
This phrase allows you to specify procedures to be performed when the READ operation is successful.

AT END Phrase
If the NEXT MODIFIED phrase is specified and there is no user-modified record in the subfile, the AT END condition exists, and the READ operation is unsuccessful.

Specify the AT END phrase when the NEXT MODIFIED phrase is used, and no applicable USE procedure is specified for the file name. If the AT END phrase and a USE procedure are both specified for a file, and the AT END condition arises, control transfers to the AT END imperative statement and the USE procedure is not run.

NOT AT END Phrase
This phrase allows you to specify procedures to be performed when the READ operation is successful.

END-READ Phrase
The END-READ phrase serves to explicitly delimit the scope of the statement.
**REWRITE Statement**

The REWRITE statement is used to replace a subfile record that already exists in the subfile.

---

**REWRITE Statement – Format 2 – TRANSACTION File (Subfile)**

```
  REWRITE SUBFILE ── record-name-1 ── FROM ── identifier-1
  FORMAT ── IS ── identifier-2 ── IS ── literal-1
  TERMINAL ── IS ── identifier-3 ── IS ── literal-2
  INDICATOR ── IS ── INDICATORS ── IS ── INDIC ── ARE ── identifier-4
  INVALID ── KEY ── imperative-statement-1
  NOT INVALID ── KEY ── imperative-statement-2 ── END-REWRITE
```

---

The number of character positions in the record referenced by record-name must be equal to the number of character positions in the record being replaced. A successful READ operation on the record must be done prior to the REWRITE operation. The record replaced in the subfile is the record in the subfile accessed by the previous READ operation.

**FORMAT Phrase**

The record format specified in the FORMAT phrase must be the record format accessed on the previous READ operation. Literal-1 or the contents of identifier-2 must be the name of the subfile format accessed on the previous READ. For more information on the FORMAT phrase, see “Common Processing Facilities” on page 181.
**TERMINAL Phrase**

The TERMINAL phrase indicates which program device’s subfile is to have a record rewritten. If the TERMINAL phrase is specified, literal-2 or identifier-3 must refer to a workstation that has been acquired by the TRANSACTION file. If literal-2 or identifier-3 contains blanks, the TERMINAL phrase has no effect. The program device specified by the TERMINAL phrase must have been acquired, either explicitly or implicitly, and must have a subfile associated with the device.

Literal-2 or identifier-3 must be a valid program device name. Literal-2, if specified, must be nonnumeric and 10 characters or fewer. Identifier-3, if specified, must refer to an alphanumeric data item, 10 characters or fewer.

If the TERMINAL phrase is omitted from a TRANSACTION file that has acquired multiple program devices, the subfile used is the subfile associated with the last program device from which a READ of the TRANSACTION file was attempted.

The REWRITE statement cannot be used for communications devices. If the REWRITE statement is used for a communications device, the operation fails and a file status of 90 is set.

**INVALID KEY Phrase**

If the RELATIVE KEY data item at the time of running the statement contains a value that does not correspond to a relative record number for the subfile, the INVALID KEY condition exists and the running of the statement is unsuccessful. To see what happens next, refer to the diagrams on pages 76 through 78.

**NOT INVALID KEY Phrase**

This phrase allows you to specify procedures to be performed when the REWRITE operation is successful.

**END-REWRI TE Phrase**

The END-REWRI TE phrase serves to explicitly delimit the scope of the statement.
WRITE Statement

The WRITE statement releases a logical record to the file.

WRITE Statement – Format 4 – TRANSACTION File (Nonsubfile)

TERMINAL Phrase

The TERMINAL phrase specifies the program devices to which the output record is to be sent.

The contents of literal-2 or identifier-3 must be the name of a program device previously acquired, either implicitly or explicitly, by the file. Literal-2, if specified, must be nonnumeric and 10 characters or fewer in length. Identifier-3, if specified, must refer to an alphanumeric data item, 10 characters or fewer in length. A value of blanks is treated as if the TERMINAL phrase were omitted.

If only a single program device was acquired by the TRANSACTION file, the
TERMINAL phrase can be omitted. That program device is always used for the WRITE.

If the TERMINAL phrase is omitted for a WRITE operation to a TRANSACTION file that has acquired multiple program devices, the default program device is used. See the discussion of the TERMINAL phrase on page 182 to see how the default program device is determined.

STARTING Phrase
The STARTING phrase specifies the starting line number for the record formats that use the variable start line keyword. This phrase is only valid for display devices.

The actual line number on which a field begins can be determined from the following equation:

\[
\text{Actual-line} = \text{Start-line} + \text{DDS Start-line} - 1
\]

*Figure 66. Line Number Equation for the STARTING Phrase*

Where:
- **Actual-line** is the actual line number
- **Start-line** is the starting line number specified in the program
- **DDS Start-line** is the line number specified in positions 39 through 41 of the Data Description Specifications form.

The WRITE operation is successful if:
- The result of the above equation is positive and less than or equal to the number of lines on the display.
- The value specified for the STARTING phrase is 0. In this case, a value of 1 is assumed.

The WRITE operation is unsuccessful, and the program ends, if:
- The result of the above equation is greater than the number of lines on the display.
- The value specified for the STARTING phrase is negative.

If the value specified for the STARTING phrase is within the screen area, any fields outside of the screen area are ignored.

Literal-3 of the STARTING phrase must be a numeric literal. Identifier-4 must be an elementary numeric item.

To use the STARTING phrase, the DDS record level keyword SLNO(*VAR) must be specified for the format being written. If the record format does not specify this keyword, the STARTING phrase is ignored at run time.

The DDS keyword CLRL also affects the STARTING phrase. CLRL controls how much of the display is cleared when the WRITE statement is processed.
See the DDS Reference for further information on SLNO(*VAR) and CLRL keywords.

ROLLING Phrase
The ROLLING phrase allows you to move lines displayed on the work station screen. All or some of the lines on the screen can be rolled up or down. The lines vacated by the rolled lines are cleared, and can have another screen format written into them. This phrase is only valid for display devices.

ROLLING is specified in the WRITE statement that is writing a new format to the display. You must specify whether the write is before or after the roll, the range of lines you want to roll, how many lines you want to roll these lines, and whether the roll operation is up or down.

After lines are rolled, the fields on these lines retain their DDS display attributes, for example, underlining, but lose their DDS usage attributes, for example, input-capability. Fields on lines that are written and then rolled (BEFORE ROLLING phrase) also lose their usage attributes.

If any part of a format is rolled, the entire format loses its usage attributes. If more than one format exists, only the rolled formats lose their usage attributes.

When you specify the ROLLING phrase, the following general rules apply.

- The DDS record level keyword ALWROL must be specified for every record format written in a WRITE statement containing the ROLLING phrase.
- Other DDS keywords mutually exclusive with the ALWROL keyword must not be used.
- Either of the DDS keywords, CLRL or OVERLAY, must be specified for a record format that is to be written and rolled to prevent the display from being cleared when that record format is written. See the DDS Reference manual for more information on DDS keywords.
- All the identifiers and literals must represent positive integer values.
- The roll starting line number (identifier-5 or literal-4) must not exceed the ending line number (identifier-6 or literal-5).
- The contents of lines that are rolled outside of the window specified by the starting and ending line numbers disappear.

Figure 67 on page 197 shows an example of a rolling operation. An initial screen format, FMT1, is written on the display. The program processes this screen format and is now ready to write the next screen format, FMT2, to the work station screen. Part of FMT1 is rolled down two lines before FMT2 is written to the display.

Processing of the following WRITE statement causes part of FMT1 to be rolled down two lines, and FMT2 to be written to the display:

```
WRITE SCREENREC FORMAT "FMT2"
   AFTER ROLLING LINES 14 THROUGH 20
   DOWN 2 LINES
```

When this WRITE statement is run, the following steps occur:

1. The contents of lines 14 through 20 are rolled down two lines.
a. The contents of lines 14 through 18 now appear on lines 16 through 20.
b. The contents of lines 14 and 15 are vacated and cleared.
c. The contents of lines 19 and 20 are rolled outside the window and disappear.

2. After the rolling operation takes place, FMT2 is written to the display.
   a. Part of FMT2 is written to the area vacated by the roll operation.
   b. Part of FMT2 is written over the data left from FMT1.

3. When the contents of the display are returned to the program by a READ statement, only the input capable fields of FMT2 are returned.
Figure 67. Example of ROLLING Operation
Format 5 can only be used for display devices. If the subfile form of the WRITE statement is used for any other type of device, the WRITE operation fails and a file status of 90 is set.

If the format is a subfile record, and SUBFILE is specified, the RELATIVE KEY clause must have been specified on the SELECT clause for the file being written. The record written to the subfile is the record in the subfile identified by the format name that has a relative record number equal to the value of the RELATIVE KEY data item. See the Data Management Guide for more information on subfiles.

**TERMINAL Phrase**

See the explanation following Format 4 for general considerations concerning the TERMINAL phrase.

The TERMINAL phrase specifies which program device’s subfile is to have a record written to it. If the TERMINAL phrase is specified, literal-2 or identifier-3 must refer to a work station associated with the TRANSACTION file. If literal-2 or identifier-3 contains a value of blanks, the TERMINAL phrase is treated as if it were not specified. The work station specified by the TERMINAL phrase must have been acquired, either explicitly or implicitly.
If the TERMINAL phrase is omitted, the subfile used is the subfile associated with
the default program device. See the discussion of the TERMINAL phrase on page
182 to see how the default program device is determined.

INVALID KEY Phrase
The INVALID KEY condition exists if a record is already in the subfile with that
record number, or if the relative record number specified is greater than the
maximum allowable subfile record number. The INVALID KEY phrase should be
specified in the WRITE SUBFILE statement for all files for which an appropriate
USE procedure is not specified.

For information about what happens when the INVALID KEY condition arises, refer
to the diagrams on pages 76 through 78.

NOT INVALID KEY Phrase
This phrase allows you to specify procedures to be performed when the WRITE
operation is successful.

END-WRITE Phrase
The END-WRITE phrase serves to explicitly delimit the scope of the statement.

For a further discussion of the WRITE statement, the FROM phrase, and the
INVALID KEY phrase, see the COBOL/400 Reference. For information on the
FORMAT phrase, see the Procedure Division, “Common Processing Facilities” on
page 181.

USE Statement
The USE statement specifies procedures for input/output error handling that are in
addition to the standard procedures provided by the input/output control system.

See the “USE Statement” section of the COBOL/400 Reference for a further dis-
cussion of the USE statement.
Examples of Work Station Programs

This section contains examples of COBOL programs that illustrate work station applications on the AS/400 system.

**Basic Inquiry Program**

Figure 68 shows the associated DDS for a basic inquiry program that uses the COBOL TRANSACTION file.

<table>
<thead>
<tr>
<th>Sequence Number</th>
<th>Condition Name</th>
<th>Length</th>
<th>Value</th>
<th>Data Type/Keyboard Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CUSTOMER MASTER INQUIRY</td>
<td>3</td>
<td>CUSTOMER NUMBER</td>
<td>R</td>
</tr>
<tr>
<td>99</td>
<td>ERRMSG ('CUSTOMER NUMBER NOT FOUND + PRESS RESET, THEN ENTER VALID NUMBER')</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>USE F3 TO END PROGRAM, USE ENTER + TO RETURN TO PROMPT SCREEN</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>A/R BALANCE</td>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The data description specifications (DDS) for the display device file (CUSMINQ) to be used by this program describe two record formats: CUSPMT and CUSFLDS.

The CUSPMT record format contains the constant ‘Customer Master Inquiry’, which identifies the display. It also contains the prompt ‘Customer Number’ and the input parameters.
field (CUST) where you enter the customer number. Five underscores appear under the input field CUST on the display where you are to enter the customer number. The error message:

Customer number not found

is also included in this record format. This message is displayed if indicator 99 is set to ON by the program. In addition, this record format defines a function key that you can press to end the program. When you press function key F3, indicator 15 is set to ON in the COBOL program. This indicator is then used to end the program.

The CUSFLDS record format contains the following constants:

- Name
- Address
- City
- State
- Zip Code
- A/R Balance.

These constants identify the fields to be written out from the program. This record format also describes the fields that correspond to these constants. All of these fields are described as output fields (blank in position 38) because they are filled in by the program; you do not enter any data into these fields. To enter another customer number, press Enter in response to this record. Notice that the CUSFLDS record overlays the CUSPMT record. Therefore, when the CUSFLDS record is written to the display, the CUSPMT record remains on the display.

In addition to describing the constants, fields, and attributes for the display, the record formats also define the line numbers and horizontal positions where the constants and fields are to be displayed.

Note: The field attributes are defined in a physical file (CUSMSTP) used for field reference purposes, instead of in the DDS for the display file. For example, EDTCDE(J) is defined in CUSMSTP for the field ARBAL.
The data description specifications (DDS) for the database file that is used by this program describe one record format: CUSMST. Each field in the record format is described, and the CUST field is identified as the key field for the record format.

Figure 69. Data Description Specification for the Record Format CUSMST.
Figure 70 (Part 1 of 2). Source Listing of a TRANSACTION Inquiry Program Using a Single Display Device.
The complete source listing for this program example is shown here. In particular, note the FILE-CONTROL and FD entries and the data structures generated by the Format 2 COPY statements.
The WRITE operation in statement 77 writes the CUSPMT format to the display. This record prompts you to enter a customer number. If you enter a customer number and press Enter, the next READ operation then reads the record back into the program.

The READ operation in statement 82 uses the customer number (CUST) field to retrieve the corresponding CUSMST record from the CUSMSTP file. If no record is found in the CUSMSTP file, indicator 99 is set on. The GO TO operation in statement 84, which is run when indicator 99 is set on, causes the program to branch back to the beginning. The message:

Customer number not found

is displayed when the format is written, because it is conditioned by indicator 99 in the DDS for the file. When you receive this message, the keyboard locks. You must press the Reset key in response to this message to unlock the keyboard. You can then enter another customer number.

If the READ operation retrieves a record from the CUSMSTP file, the WRITE operation writes the CUSFLDS record to the display work station. This record contains the customer's name, address, and accounts receivable balance.

You then press Enter, and the program branches back to the beginning. You can enter another customer number or end the program. To end the program, press F3, which sets on indicator 15 in the program.

When indicator 15 is on, the program closes all files and processes the EXIT PROGRAM statement. The program then returns control to the individual who called the COBOL program.

This is the initial display written by the WRITE operation in statement 77:

```
Customer Master Inquiry
Customer Number ________
Use F3 to end program, use enter key to return to prompt screen
```

This display appears if a record is found in the CUSMSTP file for the customer number entered in response to the first display:
Order Inquiry Programs Using Subfiles

Figure 72 on page 210 shows an example of an order inquiry program, XMPLE773, that uses subfiles. The associated DDS is also shown, except for the DDS for the customer master file, CUSMSTP. Refer to Figure 69 on page 202 for the DDS for CUSMSTP.

XMPLE773 displays all the detail order records for the requested order number. The program prompts you to enter the order number that is to be reviewed. The order number is checked against the order header file, ORDHDRP. If the order number exists, the customer number accessed from the order header file is checked against the customer master file, CUSMSTP. All order detail records in ORDDTLP for the requested order are read and written to the subfile. A write for the subfile control record format is processed, and the detail order records in the subfile are displayed for you to review. You end the program by pressing F12.
Figure 71 (Part 1 of 3). Data Description Specifications for an Order Inquiry Program
**Figure 71 (Part 2 of 3). Data Description Specifications for an Order Inquiry Program**

<table>
<thead>
<tr>
<th>Condition Name</th>
<th>Sequence Number</th>
<th>Name</th>
<th>Length</th>
<th>Location</th>
<th>Line</th>
<th>Pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item</td>
<td>0</td>
<td>Text</td>
<td>10</td>
<td>2</td>
<td>SEL</td>
<td></td>
</tr>
<tr>
<td>QtyOrd</td>
<td>3</td>
<td>Text</td>
<td>10</td>
<td>3</td>
<td>'Quantity Ordered'</td>
<td></td>
</tr>
<tr>
<td>Descr</td>
<td>30</td>
<td>Text</td>
<td>10</td>
<td>4</td>
<td>'Item Description'</td>
<td></td>
</tr>
<tr>
<td>Price</td>
<td>2</td>
<td>Text</td>
<td>10</td>
<td>5</td>
<td>'Selling Price'</td>
<td></td>
</tr>
<tr>
<td>Extens</td>
<td>2</td>
<td>Text</td>
<td>10</td>
<td>6</td>
<td>'Extension Amount Of +</td>
<td></td>
</tr>
<tr>
<td>QtyOrd X Price</td>
<td>3</td>
<td>Text</td>
<td>10</td>
<td>7</td>
<td>'Quantity Ordered X Price'</td>
<td></td>
</tr>
<tr>
<td>Extens X Price</td>
<td>2</td>
<td>Text</td>
<td>10</td>
<td>8</td>
<td>'Extension Amount Of X Price'</td>
<td></td>
</tr>
<tr>
<td>CstOrd</td>
<td>3</td>
<td>Text</td>
<td>3</td>
<td>9</td>
<td>'Customer Order Number'</td>
<td></td>
</tr>
<tr>
<td>Cname</td>
<td>20</td>
<td>Text</td>
<td>3</td>
<td>10</td>
<td>'Customer Name'</td>
<td></td>
</tr>
<tr>
<td>Addrs</td>
<td>20</td>
<td>Text</td>
<td>3</td>
<td>11</td>
<td>'Customer Address'</td>
<td></td>
</tr>
<tr>
<td>City</td>
<td>20</td>
<td>Text</td>
<td>3</td>
<td>12</td>
<td>'Customer City'</td>
<td></td>
</tr>
<tr>
<td>Stt</td>
<td>2</td>
<td>Text</td>
<td>3</td>
<td>13</td>
<td>'Customer State'</td>
<td></td>
</tr>
<tr>
<td>Zip</td>
<td>5</td>
<td>Text</td>
<td>6</td>
<td>14</td>
<td>'ZIP Code'</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>1</td>
<td>Text</td>
<td>4</td>
<td>15</td>
<td>'Total Dollar Amount Of The +</td>
<td></td>
</tr>
<tr>
<td>CstOrd</td>
<td>3</td>
<td>Text</td>
<td>3</td>
<td>16</td>
<td>'Customer Purchase Order +</td>
<td></td>
</tr>
<tr>
<td>Cntry</td>
<td>15</td>
<td>Text</td>
<td>4</td>
<td>17</td>
<td>'Customer Purchase Order +</td>
<td></td>
</tr>
<tr>
<td>Shpvia</td>
<td>15</td>
<td>Text</td>
<td>3</td>
<td>18</td>
<td>'Shipping Instructions'</td>
<td></td>
</tr>
<tr>
<td>Prntdat</td>
<td>5</td>
<td>Text</td>
<td>6</td>
<td>19</td>
<td>'Printed Date'</td>
<td></td>
</tr>
<tr>
<td>Invnum</td>
<td>5</td>
<td>Text</td>
<td>7</td>
<td>20</td>
<td>'Invoice Number'</td>
<td></td>
</tr>
<tr>
<td>Actmth</td>
<td>2</td>
<td>Text</td>
<td>4</td>
<td>21</td>
<td>'Accounting Month Of Sale'</td>
<td></td>
</tr>
<tr>
<td>Actyry</td>
<td>3</td>
<td>Text</td>
<td>7</td>
<td>22</td>
<td>'Accounting Year Of Sale'</td>
<td></td>
</tr>
<tr>
<td>Item</td>
<td>14</td>
<td>Text</td>
<td>8</td>
<td>23</td>
<td>'Item Description'</td>
<td></td>
</tr>
<tr>
<td>Price</td>
<td>48</td>
<td>Text</td>
<td>8</td>
<td>24</td>
<td>'Price'</td>
<td></td>
</tr>
<tr>
<td>Extens</td>
<td>58</td>
<td>Text</td>
<td>8</td>
<td>25</td>
<td>'Extension'</td>
<td></td>
</tr>
</tbody>
</table>
Figure 71 (Part 3 of 3). Data Description Specifications for an Order Inquiry Program

<table>
<thead>
<tr>
<th>Condition Name</th>
<th>Sequence</th>
<th>Name</th>
<th>Length</th>
<th>Location</th>
<th>Line</th>
<th>Pos</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Form Type</th>
<th>And/Or/Comment (A/O/*)</th>
<th>Not (N)</th>
<th>Indicator</th>
<th>Indicator</th>
<th>Not (N)</th>
<th>Type of Name of Spec (b/R/H/J/K/S/O)</th>
<th>Reference (R)</th>
<th>Decimal Positions</th>
<th>Usage (b/O/I/B/H/M/N/P)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

AS/400 DATA DESCRIPTION SPECIFICATIONS

Printed in U.S.A.

*Number of sheets per pad may vary slightly."
Figure 72 (Part 1 of 7). Example of an Order Inquiry Program
Chapter 8. Transaction Files

Figure 72 (Part 2 of 7). Example of an Order Inquiry Program
Figure 72 (Part 3 of 7). Example of an Order Inquiry Program
Figure 72 (Part 4 of 7). Example of an Order Inquiry Program
Figure 72 (Part 5 of 7). Example of an Order Inquiry Program
MOVE ZEROS TO ORDERN OF SUBCTL1-O.

MOVE ALL B'0' TO INDICATOR-AREA.

SET READ-DISPLAY NOT-SUBCTL1-FORMAT TO TRUE.

SET DISPLAY-SUBFILE TO TRUE.

SET NO-DETAIL-LINES-FOR-ORDER TO TRUE.

PERFORM SUBFILE-SET-UP.

SET CLEAR-SUBFILE TO TRUE.

WRITE EXISTING-ORDER-DISPLAY-RECORD FORMAT IS "SUBCTL1".

READ EXISTING-ORDER-DISPLAY-FILE RECORD.

MOVE CORR INDICATOR-AREA TO SUBCTL1-O-INDIC.

WRITE EXISTING-ORDER-DISPLAY-RECORD FORMAT IS "SUBCTL1".

READ EXISTING-ORDER-DISPLAY-FILE RECORD.

SET DISPLAY-SUBFILE CONTROL TO TRUE.
Figure 72 (Part 7 of 7). Example of an Order Inquiry Program

This is the initial order-entry prompt display written to the work station:

<table>
<thead>
<tr>
<th>Existing Order Entry</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>0000000000</td>
</tr>
<tr>
<td>Order #12400</td>
<td>Open</td>
</tr>
<tr>
<td>Date 000000</td>
<td>Customer order</td>
</tr>
<tr>
<td>Cust #</td>
<td>Ship via</td>
</tr>
<tr>
<td>000000</td>
<td>Printed date 000000</td>
</tr>
<tr>
<td>Item</td>
<td>Price</td>
</tr>
<tr>
<td>Qty</td>
<td>Extension</td>
</tr>
</tbody>
</table>

This display appears if there are detail order records for the customer whose order number was entered in the first display:
A Payment Update Program

Figure 74 on page 221 shows an example of a payment update program, PAYUPDT. For the related DDS, see Figure 73 on page 218. For the related display-screen examples, see page 228. For the DDS for the customer master file, CUSMSTP, refer to Figure 69 on page 202.

In this example, payments from customers are registered. The clerk is prompted to enter one or more customer numbers and the amount of money to be credited to each customer’s account. The program checks the customer number and unconditionally accepts any payment for an existing customer who has invoices outstanding. If an overpayment will result from the amount of the payment from a customer, the clerk is given the option to accept or reject the payment. If no customer record exists for a customer number, an error message is issued. Payments can be entered until the clerk ends the program by pressing F12.
Figure 73 (Part 1 of 3). Example of a Data Description Specification for a Payment Update Program
### AS/400 DATA DESCRIPTION SPECIFICATIONS

#### Figure 73 (Part 2 of 3). Example of a Data Description Specification for a Payment Update Program

<table>
<thead>
<tr>
<th>Condition Name</th>
<th>Sequence Number</th>
<th>Length</th>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS FOR THE DISPLAY DEVICE FILE PAYUPD1D</td>
<td>1</td>
<td></td>
<td></td>
<td>ACCOUNTS RECEIVABLE INTERACTIVE PAYMENT UPDATE</td>
</tr>
<tr>
<td>R SUBFILE1</td>
<td>2</td>
<td></td>
<td></td>
<td>SFL</td>
</tr>
<tr>
<td>ACCOUNTS RECEIVABLE INTERACTIVE PAYMENT UPDATE</td>
<td>3</td>
<td></td>
<td></td>
<td>TEXT('SUBFILE FOR CUSTOMER PAYMENT')</td>
</tr>
<tr>
<td>ACPPMT</td>
<td>4</td>
<td>5</td>
<td></td>
<td>TEXT('ACCEPT PAYMENT')</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td>VALUES('YES', 'NO')</td>
</tr>
<tr>
<td>51</td>
<td>6</td>
<td></td>
<td></td>
<td>DSPATR(R I MDT)</td>
</tr>
<tr>
<td>N51</td>
<td>7</td>
<td></td>
<td></td>
<td>DSPATR(ND PR)</td>
</tr>
<tr>
<td>CUST</td>
<td>8</td>
<td>9</td>
<td></td>
<td>TEXT('CUSTOMER NUMBER')</td>
</tr>
<tr>
<td>52</td>
<td>10</td>
<td></td>
<td></td>
<td>DSPATR(R I)</td>
</tr>
<tr>
<td>53</td>
<td>11</td>
<td></td>
<td></td>
<td>DSPATR(ND)</td>
</tr>
<tr>
<td>54</td>
<td>12</td>
<td></td>
<td></td>
<td>DSPATR(PR)</td>
</tr>
<tr>
<td>AMPAID</td>
<td>13</td>
<td>14</td>
<td></td>
<td>TEXT('AMOUNT PAID')</td>
</tr>
<tr>
<td>8028</td>
<td>15</td>
<td></td>
<td></td>
<td>CHECK(FE)</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td></td>
<td></td>
<td>AUTO(RAB)</td>
</tr>
<tr>
<td>52</td>
<td>17</td>
<td></td>
<td></td>
<td>CMP(GT O )</td>
</tr>
<tr>
<td>53</td>
<td>18</td>
<td></td>
<td></td>
<td>DSPATR(R I)</td>
</tr>
<tr>
<td>54</td>
<td>19</td>
<td></td>
<td></td>
<td>DSPATR(ND)</td>
</tr>
<tr>
<td>ECPMSG</td>
<td>20</td>
<td>21</td>
<td></td>
<td>TEXT('EXCEPTION MESSAGE')</td>
</tr>
<tr>
<td>31A0</td>
<td>22</td>
<td></td>
<td></td>
<td>DSPATR(R I)</td>
</tr>
<tr>
<td>53</td>
<td>23</td>
<td></td>
<td></td>
<td>DSPATR(ND)</td>
</tr>
<tr>
<td>54</td>
<td>24</td>
<td></td>
<td></td>
<td>DSPATR(PR)</td>
</tr>
<tr>
<td>OVRPMT</td>
<td>25</td>
<td>26</td>
<td></td>
<td>TEXT('OVERPAYMENT')</td>
</tr>
<tr>
<td>8Y20</td>
<td>27</td>
<td></td>
<td></td>
<td>EDTCDE(1)</td>
</tr>
<tr>
<td>56</td>
<td>28</td>
<td></td>
<td></td>
<td>DSPATR(BL)</td>
</tr>
<tr>
<td>N56</td>
<td>29</td>
<td></td>
<td></td>
<td>DSPATR(ND)</td>
</tr>
<tr>
<td>STSCDE</td>
<td>30</td>
<td></td>
<td></td>
<td>TEXT('STATUS CODE')</td>
</tr>
</tbody>
</table>

---

Chapter 8. Transaction Files 219
### AS/400 DATA DESCRIPTION SPECIFICATIONS

**Figure 73 (Part 3 of 3). Example of a Data Description Specification for a Payment Update Program**

<table>
<thead>
<tr>
<th>Description</th>
<th>Page of</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Data Description Specifications</td>
<td>Description</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Condition Name</th>
<th>Sequence Number</th>
<th>Name</th>
<th>Length</th>
<th>Location</th>
<th>Line</th>
<th>Pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>R CONTROL1</td>
<td>TEXT('SUBFILE CONTROL')</td>
<td>SFLCTL(SUBFILE1)</td>
<td>SFLSIZ(17)</td>
<td>SFLPAG(17)</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>61</td>
<td>SFLCLR</td>
<td>SFDSP</td>
<td>SFLDSPCTL</td>
<td>OVERLAY</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>62</td>
<td>LOCK</td>
<td>HELP(99 'HELP KEY')</td>
<td>CA12(98 'END PAYMENT UPDATE')</td>
<td>CA11(97 'IGNORE INPUT')</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>99</td>
<td>SFLMSG('F11 - IGNORE INVALID INPUT; F12 - END PAYMENT UPDATE')</td>
<td>A</td>
<td>12</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R MESSAGE1</td>
<td>TEXT('MESSAGE RECORD')</td>
<td>OVERLAY</td>
<td>LOCK</td>
<td>15</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>24 2*ACCEPT PAYMENT VALUES: (*<em>NO <em>YES)</em></em></td>
<td>DSPATR(R1)</td>
<td>18</td>
<td>19</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**International Business Machines**

**File**

A CONTROL 1 TEXT('SUBFILE CONTROL')
A SF LCTL(SUBFILE1)
A SFLSIZ(17)
A SFLPAG(17)
A SFLCLR
A SFDSP
A SFLDSPCTL
A OVERLAY
A LOCK
A HELP(99 'HELP KEY')
A CA12(98 'END PAYMENT UPDATE')
A CA11(97 'IGNORE INPUT')
A SFLMSG('F11 - IGNORE INVALID INPUT; F12 - END PAYMENT UPDATE')
A R MESSAGE1 TEXT('MESSAGE RECORD')
A OVERLAY
A LOCK
A 71 24 2*ACCEPT PAYMENT VALUES: (**NO *YES)**
A DSPATR(R1)
Figure 74 (Part 1 of 8). Source Listing of a Payment Update Program Example
Figure 74 (Part 2 of 8). Source Listing of a Payment Update Program Example
Chapter 8. Transaction Files

Figure 74 (Part 3 of 8). Source Listing of a Payment Update Program Example

```cobol
5763CB1 V3R0M5 AS/400 COBOL Source
STMT SEQNR - A 1 B..+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME CHG DATE
95 +000033 06 OVRMT PIC S9(6)V9(2). <-ALL-FMTS
96 +000034+ OVRPAYMT PIC X(1). <-ALL-FMTS
+000036+ STSCDE PIC X(1). <-ALL-FMTS
+000037+ STATUS CODE <-ALL-FMTS
+000038+ INPUT FORMAT:CONTROL1 FROM FILE PAYUPDTD OF LIBRARY XMPLIB <-ALL-FMTS
97 +000039 05 CONTROL-I REDEFINES PAYUPDTD-RECORD. <-ALL-FMTS
98 +000040 06 CONTROL-I-INDIC. <-ALL-FMTS
99 +000041 07 IN99 PIC 1 INDIC 99. <-ALL-FMTS
+000042+ HELP KEY <-ALL-FMTS
100 +000043 07 IN98 PIC 1 INDIC 98. <-ALL-FMTS
+000044+ END PAYMENT UPDATE <-ALL-FMTS
101 +000045 07 IN97 PIC 1 INDIC 97. <-ALL-FMTS
+000046+ OUTPUT FORMAT:CONTROL1 FROM FILE PAYUPDTD OF LIBRARY XMPLIB <-ALL-FMTS
+000047+ IGNORE INPUT <-ALL-FMTS
+000048+ SUBFILE CONTROL <-ALL-FMTS
102 +000049 05 CONTROL-D REDEFINES PAYUPDTD-RECORD. <-ALL-FMTS
103 +000050 06 CONTROL-D-INDIC. <-ALL-FMTS
104 +000051 07 IN61 PIC 1 INDIC 61. <-ALL-FMTS
105 +000052 07 IN62 PIC 1 INDIC 62. <-ALL-FMTS
106 +000053 07 IN99 PIC 1 INDIC 99. <-ALL-FMTS
+000054+ HELP KEY <-ALL-FMTS
107 +000055 07 IN63 PIC 1 INDIC 63. <-ALL-FMTS
108 +000056 07 IN64 PIC 1 INDIC 64. <-ALL-FMTS
109 +000057+ INPUT FORMAT:MESSAGE1 FROM FILE PAYUPDTD OF LIBRARY XMPLIB <-ALL-FMTS
+000058+ MESSAGE RECORD <-ALL-FMTS
+000059+ MESSAGE RECORD <-ALL-FMTS
+000060+ OUTPUT FORMAT:MESSAGE1 FROM FILE PAYUPDTD OF LIBRARY XMPLIB <-ALL-FMTS
+000061+ MESSAGE RECORD <-ALL-FMTS
109 +000062 05 MESSAGE-D REDEFINES PAYUPDTD-RECORD. <-ALL-FMTS
110 +000063 06 MESSAGE-D-INDIC. <-ALL-FMTS
111 +000064 07 IN71 PIC 1 INDIC 71. <-ALL-FMTS
004500
112 004600 WORKING-STORAGE SECTION.
004700
113 004800 01 REL-NUMBER PIC 9(05)
114 004900 VALUE ZEROS.
005000
115 005100 01 WS-CONTROL.
116 005200 05 WS-IND PIC X(02).
117 005300 05 WS-FORMAT PIC X(10).
118 005400 01 SYSTEM-DATE.
119 005500 05 SYSTEM-YEAR PIC 99.
120 005600 05 SYSTEM-MONTH PIC 99.
121 005700 05 SYSTEM-DAY PIC 99.
122 005800 01 PROGRAM-DATE.
123 005900 05 PROGRAM-MONTH PIC 99.
124 006000 05 PROGRAM-DAY PIC 99.
125 006100 05 PROGRAM-YEAR PIC 99.
126 006200 01 FILE-DATE REDEFINES PROGRAM-DATE
127 006300 PIC S9(6).
128 006400 01 EXCEPTION-STATUS.
129 006500 05 STATUS-CODE-ONE PIC XX.
130 006600 88 SUBFILE-IS-FULL VALUE '9M'.
131 006700 01 EXCEPTION-MESSAGES.
132 006800 05 MESSAGE-ONE PIC X(31)
133 006900 VALUE 'CUSTOMER DOES NOT EXIST'.
134 007000 05 MESSAGE-TWO PIC X(31)
135 007100 VALUE 'NO INVOICES EXIST FOR CUSTOMER'.
136 007200 05 MESSAGE-THREE PIC X(31)
137 007300 VALUE 'CUSTOMER HAS AN OVER PAYMENT OF'.
138 007400 01 PROGRAM-VARIABLES.
139 007500 05 AMOUNT-DQED PIC S9(6)V99.
140 007600 05 AMOUNT-PAYD PIC S9(6)V99.
141 007700 05 INVOICE-BALANCE PIC S9(6)V99.
142 007800 01 ERRPMT-PARAMETERS.
143 007900 05 DISPLAY-PARAMETER PIC X(8)
144 008000 VALUE 'PAYUPDTD'.
145 008100 05 DUMMY-ONE PIC X(6)
146 008200 VALUE SPACES.
147 008300 05 DUMMY-TWO PIC X(6)
148 008400 VALUE SPACES.
149 008500 05 STATUS-CODE-TWO.
150 008600 10 PRIMARY PIC X(1).
151 008700 10 SECONDARY PIC X(1).
```
224 COBOL/400 User's Guide

Figure 74 (Part 4 of 8). Source Listing of a Payment Update Program Example
PROCEDURE DIVISION.

IDENTIFICATION SECTION.
COPYNAME CHG DATE

PROCEDURE DIVISION.

DECLARATIVES.

TRANSACTION-ERROR SECTION.
USE AFTER STANDARD ERROR PROCEDURE
PAYMENT-UPDATE-DISPLAY-FILE.
WORK-STATION-ERROR-HANDLER.

IF SUBFILE-IS-FULL THEN
NEXT SENTENCE
ELSE
DISPLAY 'ERROR IN PAYMENT-UPDATE' STATUS-CODE-ONE.
END DECLARATIVES.

CUSTOMER-PAYMENT-UPDATE SECTION.
MAINLINE-ROUTINE.

PERFORM SET-UP-ROUTINE.
PERFORM PROCESS-TRANSACTION-FILE UNTIL END-OF-PAYMENT-UPDATE.
PERFORM CLEAN-UP-ROUTINE.

OPEN I-O CUSTOMER-INVOICE-FILE CUSTOMER-MASTER-FILE PAYMENT-UPDATE-DISPLAY-FILE.
MOVE ALL B' ' TO INDICATOR-AREA SWITCH-AREA.
ACCEPT SYSTEM-DATE FROM DATE.
MOVE SYSTEM-YEAR TO PROGRAM-YEAR.
MOVE SYSTEM-MONTH TO PROGRAM-MONTH.
MOVE SYSTEM-DAY TO PROGRAM-DAY.
CLEAR-SUBFILE TO TRUE.
MOVE CORR INDICATOR-AREA TO CONTROL1-O-INDIC.
WRITE PAYMENT-UPDATE-DISPLAY-RECORD FORMAT IS 'CONTROL1'.
SET DO-NOT-CLEAR-SUBFILE TO TRUE.
PERFORM INITIALIZE-SUBFILE-RECORD 17 TIMES.
MOVE CORR CONTROL1-I-INDIC TO INDICATOR-AREA.
PROCESS-TRANSACTION-FILE.

IF HELP-IS-NOT-NEEDED THEN
IF IGNORE-INPUT THEN
MOVE CORR INDICATOR-AREA TO CONTROL1-O-INDIC.
WRITE PAYMENT-UPDATE-DISPLAY-RECORD FORMAT IS 'CONTROL1'.
SET DO-NOT-CLEAR-SUBFILE TO TRUE.
PERFORM INITIALIZE-SUBFILE-RECORD 17 TIMES.
ELSE
MOVE CORR INDICATOR-AREA TO CONTROL1-O-INDIC.
WRITE PAYMENT-UPDATE-DISPLAY-RECORD FORMAT IS 'CONTROL1'.
SET DO-NOT-CLEAR-SUBFILE TO TRUE.
PERFORM INITIALIZE-SUBFILE-RECORD 17 TIMES.
ELSE
PERFORM READ-MODIFIED-SUBFILE-RECORD.

Figure 74 (Part 5 of 8). Source Listing of a Payment Update Program Example
Figure 74 (Part 6 of 8). Source Listing of a Payment Update Program Example
5763CB1 V3R/zerodotM5 AS/4/zerodot/zerodot COBOL Source
STMT SEQNBR -A 1 B..+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME CHG DATE
297 031500 PERFORM PAYMENT-UPDATE
298 031600 UNTIL NO-MORE-INVOICES-EXIST
299 031700 OR NO-MORE-PAYMENT-EXIST
300 031800 IF ARBAL OF CUSTOMER-MASTER-RECORD IS NEGATIVE
301 031900 SET DO-NOT-DISPLAY-FIELD
302 032000 DO-NOT-DISPLAY-ACCEPT-PAYMENT
303 032100 PROTECT-INPUT-FIELD TO TRUE
304 032200 MOVE ARBAL TO OVRPMT OF SUBFILE1-O
305 032300 MOVE MESSAGE-THREE TO ECPMSG OF SUBFILE1-O
306 032400 MOVE '/zerodot' TO STSCDE OF SUBFILE1-O
307 032500 PERFORM REWRITE-DISPLAY-SUBFILE-RECORD
308 032600 ELSE
309 032700 SET DO-NOT-DISPLAY-accept
310 032800 SET DO-NOT-DISPLAY-OVER-PAYMENT
311 032900 SET DO-NOT-REVERSE-FIELD-IMAGE
312 033000 MOVE ZEROS TO OVRPMT OF SUBFILE1-O
313 033100 MOVE MESSAGE-ONE TO ECPMSG OF SUBFILE1-O
314 033200 MOVE '1' TO STSCDE OF SUBFILE1-O
315 033300 PERFORM REWRITE-DISPLAY-SUBFILE-RECORD.
316 033400 ELSE
317 033500 PERFORM NO-CUSTOMER-INVOICE-ROUTINE
318 033600 ELSE
319 033700 SET REVERSE-FIELD-IMAGE
320 033800 DO-NOT-PROTECT-INPUT-FIELD
321 033900 DISPLAY-FIELD
322 034000 DO-NOT-DISPLAY-OVER-PAYMENT
323 034100 DO-NOT-MAKE-FIELD-BLINK
324 034200 DO-NOT-REVERSE-FIELD-IMAGE
325 034300 DO-NOT-MAKE-FIELD-BLINK
326 034400 DO-NOT-REVERSE-FIELD-IMAGE
327 034500 DO-NOT-DISPLAY-ACCEPT-PAYMENT
328 034600 PROTECT-INPUT-FIELD TO TRUE
329 034700 MOVE AMPAID OF SUBFILE1-I TO LSTAMT OF CUSTOMER-MASTER-RECORD.
330 034800 MOVE '/zerodot' TO STSCDE OF SUBFILE1-O
331 034900 PERFORM REWRITE-DISPLAY-SUBFILE-RECORD.
332 035000 IF STSCDE OF SUBFILE1-O IS EQUAL TO '1' THEN
333 035100 MOVESUBFILEPAYMENT-UPDATE-DISPLAY-RECORD
334 035200 FORMAT IS 'SUBFILE1'.
335 035300 ELSE
336 035400 SET DO-NOT-DISPLAY-FIELD

Figure 74 (Part 7 of 8). Source Listing of a Payment Update Program Example
This is the initial display that is written to the work station to prompt you to enter the customer number and payment:
Enter the customer numbers and payments:

<table>
<thead>
<tr>
<th>Customer</th>
<th>Payment</th>
</tr>
</thead>
<tbody>
<tr>
<td>34500</td>
<td>2000</td>
</tr>
<tr>
<td>40500</td>
<td>30000</td>
</tr>
<tr>
<td>36000</td>
<td>2500</td>
</tr>
<tr>
<td>12500</td>
<td>200</td>
</tr>
<tr>
<td>22799</td>
<td>4500</td>
</tr>
<tr>
<td>41900</td>
<td>7500</td>
</tr>
<tr>
<td>10001</td>
<td>5000</td>
</tr>
<tr>
<td>49500</td>
<td>2500</td>
</tr>
<tr>
<td>13300</td>
<td>3500</td>
</tr>
<tr>
<td>56900</td>
<td>4000</td>
</tr>
</tbody>
</table>

Payments that would result in overpayments or that have incorrect customer numbers are left on the display and appropriate messages are added:
Customer Payment Update Prompt

<table>
<thead>
<tr>
<th>Accept Payment</th>
<th>Customer</th>
<th>Payment</th>
<th>Exception Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>40500</td>
<td>30000</td>
<td>NO INVOICES EXIST FOR CUSTOMER</td>
</tr>
<tr>
<td>NO</td>
<td>12500</td>
<td>200</td>
<td>NO INVOICES EXIST FOR CUSTOMER</td>
</tr>
<tr>
<td>NO</td>
<td>41900</td>
<td>7500</td>
<td>NO INVOICES EXIST FOR CUSTOMER</td>
</tr>
<tr>
<td>NO</td>
<td>10001</td>
<td>5000</td>
<td>CUSTOMER DOES NOT EXIST</td>
</tr>
<tr>
<td>NO</td>
<td>13300</td>
<td>3500</td>
<td>NO INVOICES EXIST FOR CUSTOMER</td>
</tr>
</tbody>
</table>

Accept payment values: (+NO +YES)

Indicate which payments to accept:

Accept payment values: (+NO +YES)

The accepted payments are processed, and overpayment information is displayed:
<table>
<thead>
<tr>
<th>Customer</th>
<th>Payment</th>
<th>Exception Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>12500</td>
<td>200</td>
<td>CUSTOMER HAS AN OVERPAYMENT OF 58.50</td>
</tr>
<tr>
<td>10001</td>
<td>5000</td>
<td>CUSTOMER DOES NOT EXIST</td>
</tr>
</tbody>
</table>
Chapter 9. Printer Files

This chapter describes how COBOL/400 interacts with the different kinds of printer files.

You can obtain printed output from a COBOL program by issuing WRITE statements to one or more printer files. Each printer file must have a unique name and be assigned to a device of PRINTER or FORMATFILE in the ASSIGN clause of that file’s FILE-CONTROL entry.

A device of PRINTER must be used for program-described files, and a device of FORMATFILE must be used for externally described printer files. The Create Print File (CRTPRTF) command can be used to create a printer file (see the CL Reference for further information on the CRTPRTF command), or one of the IBM-supplied printer-device files, such as QPRINT can be used.

The file operations that are valid for a printer file are WRITE, OPEN, and CLOSE. For a complete description of these operations, see the COBOL/400 Reference.

See the DDS Reference for information on the DDS for externally described printer files. For more information on FORMATFILE files, see “FORMATFILE Files” on page 234.

SPECIAL-NAMES Paragraph and the ADVANCING Phrase

When the mnemonic-name associated with the function-name CSP is specified in the ADVANCING phrase of a WRITE statement for a printer file, it has the same effect as specifying ADVANCING 0 LINES.

When the mnemonic-name associated with the function-name C01 is specified in the ADVANCING phrase of a WRITE statement for a printer file, it has the same effect as specifying ADVANCING PAGE.

The ADVANCING phrase cannot be specified in WRITE statements for files assigned to FORMATFILE.

LINAGE Clause

When the LINAGE clause is specified for a file assigned to PRINTER, all spacing and paging controls are handled internally by compiler generated code.

Paper positioning is done only when the first WRITE statement is run. The paper in the printer is positioned to a new physical page, and the LINAGE-COUNTER is set to 1. When the printer file is shared and other programs have written records to the file, the COBOL WRITE statement is still considered to be the first WRITE statement. Paper positioning is handled by the COBOL/400 compiler even though it is not the first WRITE statement for that file.

All spacing and paging for WRITE statements is controlled internally. The physical size of the page is ignored when paper positioning is not properly defined for the COBOL/400 compiler. For a file that has a LINAGE clause and is assigned to PRINTER, paging consists of spacing to the end of the logical page (page body) and then spacing past the bottom and top margins.
Use of the LINAGE clause degrades performance. The LINAGE clause should be used only as necessary. If the physical paging is acceptable, the LINAGE clause is not necessary.

The LINAGE clause should not be used for files assigned to FORMATFILE.

### FORMATFILE Files

Externally described printer files must be assigned to a device of FORMATFILE. The term FORMATFILE is used because the FORMAT phrase is valid in WRITE statements for the file, and the data formatting is specified in the DDS for the file.

When you have specified a device of FORMATFILE, you can obtain formatting of printed output in two ways:

1. Choose the formats to print and their order by using appropriate values in the FORMAT phrases specified for WRITE statements. For example, use one format once per page to produce a heading, and use another format to produce the detail lines on the page.

2. Choose the appropriate options to be taken when each format is printed by setting indicator values and passing these indicators through the INDICATOR phrase for the WRITE statement. For example, fields may be underlined, blank lines may be produced before or after the format is printed, or the printing of certain fields may be skipped.

The use of external descriptions for printer files has the following advantages over program descriptions:

- Multiple lines can be printed by one WRITE statement. When multiple lines are written by one WRITE statement and the END-OF-PAGE condition is reached, the END-OF-PAGE imperative statement is processed after all of the lines are printed. It is possible to print lines in the overflow area, and onto the next page before the END-OF-PAGE imperative statement is processed.

  Figure 75 on page 235 shows an example of an occurrence of the END-OF-PAGE condition through FORMATFILE.

- Optional printing of fields based on indicator values is possible.

- Editing of field values is easily defined.

- Maintenance of print formats, especially those used by multiple programs, is easier.

Use of the ADVANCING phrase for FORMATFILE files causes a compilation error to be issued. Advancing of lines is controlled in a FORMATFILE file through DDS keywords, such as SKIPA and SKIPB, and through the use of line numbers.

For FORMATFILE files, the LINAGE clause is invalid.
Figure 75 (Part 1 of 2). Example of the END-OF-PAGE Condition
The externally described printer file is assigned to device FORMATFILE.

The Format 2 COPY statement is used to copy the fields for the printer file into the program.

Note that, although the fields in format DETAIL will be printed on three separate lines, they are defined in one record.

COPY-DDS is used to copy the indicators used in the printer file into the program.

Paragraph PROCESS-RECORD processes PRINT-RECORD for each employee record.

All fields in the employee record are moved to the record for format DETAIL.

If the employee is married, indicator 01 is turned on; if not, the indicator is turned off, preventing the spouse’s name field in DETAIL from being printed.

Format DETAIL is printed with indicator 01 passed to control printing.

If the number of lines per page has been exceeded, END-OF-PAGE occurs. The format HEADING is printed on a new page.
### DDS Example of the Use of Externally Described Printer Files Assigned to a Device of FORMATFILE

**Figure 76 (Part 1 of 2)**

```
A
A  R PERSREC
A  ENPLNG  65
A  NAME   30
A  ADDRESS1 35
A  ADDRESS2 20
A  BIRTHDATE 6
A  MARSTAT  1
A  SPOUSENAME 30
A  NUMCHILD  25
A  K ENPLNG
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
```

*AS/400 DATA DESCRIPTION SPECIFICATIONS*

*Number of pages per pad may vary.*

*Printed in U.S.A.*
Figure 76 (Part 2 of 2). DDS Example of the Use of Externally Described Printer Files Assigned to a Device of FORMATFILE

1. INDARA specifies that a separate indicator area is to be used for the file.

2. HEADING is the format name that provides headings for each page.

3. SKIPB(1) and SPACEA(3) are used to:
   1. Skip to line 1 of the next page before format HEADING is printed.
   2. Leave 3 blank lines after format HEADING is printed.

4. DATE, TIME, and PAGNBR are used to have the current date, time and page number printed automatically when format HEADING is printed.

5. DETAIL is the format name used to print the detail line for each employee in the personnel file.
SPACEA(3) causes three lines to be left blank after each employee detail line.

SPACEA(1) causes a blank line to be printed after the field BIRTHDATE is printed. As a result, subsequent fields in the same format are printed on a new line.

01 means that these fields are printed only if the COBOL program turns indicator 01 on and passes it when format DETAIL is printed.

EDTCDE(3) is used to remove leading zeros when printing this numeric field.
Chapter 10. DISK and DATABASE Files

Database files, which are associated with the COBOL devices of DATABASE and DISK, can be:

- Externally described files, whose fields are described to OS/400 through DDS
- Program-described files, whose fields are described in the program that uses the file.

All database files are created by OS/400 Create File commands. See the Database Guide for a description of the Create File commands for database files.

DATABASE versus DISK Files

Assigning a file to DISK in COBOL restricts the user to traditional DISK processing. The use of DATABASE as the device permits the user to make use of the special COBOL/400 database features such as formats and duplicate record keys.

Processing Methods for DISK and DATABASE Files

COBOL Indexed Files

An indexed file is a file whose access path is built on key values. The user must create a keyed access path for an indexed file by using DDS.

To write standard ANSI X3.23-1985 COBOL programs that access an indexed file, you must create the file with certain characteristics. The following table lists these characteristics and what controls them:

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>The file must be a physical file.</td>
<td>The CL command CRTPF</td>
</tr>
<tr>
<td>The file cannot have records with duplicate key values.</td>
<td>The DDS keyword UNIQUE</td>
</tr>
<tr>
<td>The file cannot be a shared file.</td>
<td>The CL command CRTPF</td>
</tr>
<tr>
<td>A key must be defined for the file.</td>
<td>DDS</td>
</tr>
<tr>
<td>Keys must be in ascending sequence.</td>
<td>DDS</td>
</tr>
<tr>
<td>Keys must be contiguous within the record.</td>
<td>DDS</td>
</tr>
<tr>
<td>Key fields must be alphanumeric. They cannot be numeric only.</td>
<td>DDS</td>
</tr>
<tr>
<td>The value of the key used for sequencing must include all 8 bits of every byte.</td>
<td>DDS</td>
</tr>
<tr>
<td>A starting position for retrieving records cannot be specified.</td>
<td>The CL command OVRDBF</td>
</tr>
<tr>
<td>Select/omit level keywords cannot be used for the file.</td>
<td>DDS</td>
</tr>
</tbody>
</table>

An indexed file is identified by the ORGANIZATION IS INDEXED clause of the SELECT statement.
The key fields identify the records in an indexed file. The user specifies the key field in the RECORD KEY clause of the SELECT statement. The RECORD KEY data item must be defined within a record description for the indexed file. If there are multiple record descriptions for the file, only one need contain the RECORD KEY data name. However, the same positions within the record description that contain the RECORD KEY data item are accessed in the other record descriptions as the KEY value for any references to the other record descriptions for that file.

An indexed file can be accessed sequentially, randomly by key, or dynamically.

**Valid RECORD KEYS**

The DDS for the file specifies the fields to be used as the key field. If the file has multiple key fields, the key fields must be contiguous in each record unless RECORD KEY IS EXTERNALLY-DESCRIBED-KEY is specified.

When the DDS specifies only one key field for the file, the RECORD KEY must be a single field of the same length as the key field defined in the DDS.

If a Format 2 COPY statement is specified for the file, the RECORD KEY clause must specify one of the following:

- The name used in the DDS for the key field if the name is not a COBOL reserved word.
- The name used in the DDS for the key field with -DDS added to the end if the name is a COBOL reserved word.
- The data name defined with the proper length and at the proper location in a program-described record description for the file.
- EXTERNALLY-DESCRIBED-KEY. This keyword specifies that the keys defined in DDS for each record format are to be used for accessing the file. These keys can be noncontiguous. They can be defined at different positions within the record format.

When the DDS specifies multiple contiguous key fields, the RECORD KEY data name must be a single field with its length equal to the sum of the lengths of the multiple key fields in the DDS. If a Format 2 COPY statement is specified for the file, there must also be a program-described record description for the file that defines the RECORD KEY data name with the proper length and at the proper position in the record.

**Contiguous items** are consecutive elementary or group items in the Data Division that are contained in a single data hierarchy.

**Referring to a Partial Key**

A generic START statement allows the use of a partial key. The KEY IS phrase is required.

Refer to the “START Statement” in the COBOL/400 Reference for information about the rules for specifying a search argument that refers to a partial key.

Figure 77 on page 243 shows an example of generic START statements using a program-described file.
Figure 77. Generic START Statements Using a Program-Described File

Figure 78 and Figure 79 show an example of generic START statements using an externally described file.
<table>
<thead>
<tr>
<th>SEQNR</th>
<th>DATA DESCRIPTION SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>A RREDE UNIQUE</td>
</tr>
<tr>
<td>200</td>
<td>A FNAME 20 TEXT('FIRST NAME')</td>
</tr>
<tr>
<td>300</td>
<td>A MINAME 1 TEXT('MIDDLE INITIAL NAME')</td>
</tr>
<tr>
<td>400</td>
<td>A MNAME 19 TEXT('REST OF MIDDLE NAME')</td>
</tr>
<tr>
<td>500</td>
<td>A LNAME 20 TEXT('LAST NAME')</td>
</tr>
<tr>
<td>600</td>
<td>A PHONE 10 0 TEXT('PHONE NUMBER')</td>
</tr>
<tr>
<td>700</td>
<td>A DATA 40 TEXT('REST OF DATA')</td>
</tr>
<tr>
<td>900</td>
<td>A K LNAME</td>
</tr>
<tr>
<td>1000</td>
<td>A K FNAME</td>
</tr>
<tr>
<td>1100</td>
<td>A K MINAME</td>
</tr>
<tr>
<td>1200</td>
<td>A K MNAME</td>
</tr>
</tbody>
</table>

Figure 78. Generic START Statements Using an Externally Described File -- DDS
Figure 79. Generic START Statements Using an Externally Described File
Logical File Considerations

When a logical file with multiple record formats, each having associated key fields, is processed as an indexed file in COBOL, the following restrictions and considerations apply:

- The FORMAT phrase must be specified on all WRITE statements to the file unless a Record Format Selector Program exists and has been specified in the FMTSLR parameter of the Create Logical File (CRTLF) command, the Change Logical File (CHGLF) command, or the Override Database File (OVRDBF) command. For information on the use of format selector programs, refer to the Database Guide.

- If the access mode is RANDOM or DYNAMIC, and the DUPLICATES phrase is not specified for the file, the FORMAT phrase must be specified on all DELETE and REWRITE statements.

- When the FORMAT phrase is not specified, only the portion of the RECORD KEY data item that is common to all record formats for the file is used by the system as the key for the I/O statement. When the FORMAT phrase is specified, only the portion of the RECORD KEY data item that is defined for the specified record format is used by the system as the key. See the Database Guide for more information on logical file processing.

- When *NONE is specified as the first key field for any format in a file, records can only be accessed sequentially. When a file is read randomly:
  - If a format name is specified, the first record with the specified format is returned.
  - If a format name is not specified, the first record in the file is returned.

In both cases, the value of the RECORD KEY data item is ignored.

- For a program-defined key field:
  - Key fields within each record format must be contiguous.
  - The first key field for each record format must begin at the same relative position within each record.
  - The length of the RECORD KEY data item must be equal to the length of the longest key for any format in the file.

- For an EXTERNALLY-DESCRIBED-KEY:
  - Key fields within each record format can be noncontiguous.
  - The key fields can begin at different positions in each record format.

Figure 80 on page 247 and Figure 81 on page 248 show examples of how to use DDS to describe the access path for indexed files.
Data description specifications must be used to create the access path for a program-described indexed file.

In the DDS for the record format FORMATA for the logical file ORDDTLL, the field ORDERN, which is five digits long, is defined as the key field. The definition of ORDERN as the key field establishes the keyed access for this file. Two other fields, FLDA and FLDB, describe the remaining positions in this record as character fields.

The program-described input field ORDDTLL is described in the FILE-CONTROL section in the SELECT clause as an indexed file.
The COBOL descriptions of each field in the FD entry must agree with the corresponding description in the DDS file. The RECORD KEY data item must be defined as a five-digit numeric integer beginning in position 15 of the record.

<table>
<thead>
<tr>
<th>Condition Name</th>
<th>Sequence Number</th>
<th>Name</th>
<th>Length</th>
<th>Location</th>
<th>Line</th>
<th>Pos</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram](image)

In this example, the data description specifications define two key fields for the record format FORMAT in the logical file ORDDTLL. For the two fields to be used as a composite key for a program-described indexed file, the key fields must be contiguous in the record.

The COBOL description of each field must agree with the corresponding description in the DDS file. A 10-character item beginning in position 15 of the record must be defined in the RECORD KEY clause of the file-control entry. The COBOL
descriptions of the DDS fields ORDERN and ITEM would be subordinate to the 10-character item defined in the RECORD KEY clause.

**COBOL Relative Files**

A COBOL relative file is a file to be processed by a relative record number. To process a file by relative record number, you must specify ORGANIZATION IS RELATIVE in the SELECT statement for the file. A relative file can be accessed sequentially, randomly by record number, or dynamically.

To write standard ANSI X3.23-1985 COBOL programs that access a relative file, you must create the file with certain characteristics. The following table lists these characteristics and what controls them.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>The file must be a physical file.</td>
<td>The CL command CRTPF</td>
</tr>
<tr>
<td>The file cannot be a shared file.</td>
<td>The CL command CRTPF</td>
</tr>
<tr>
<td>No key can be specified for the file.</td>
<td>DDS</td>
</tr>
<tr>
<td>A starting position for retrieving records cannot be specified.</td>
<td>The CL command OVRDBF</td>
</tr>
<tr>
<td>Select/omit level keywords cannot be used for the file.</td>
<td>DDS</td>
</tr>
<tr>
<td>Records in the file cannot be reused.</td>
<td>The CL command CRTPF</td>
</tr>
</tbody>
</table>

For a COBOL file with an organization of RELATIVE, the Reorganize Physical File Member (RGZPFM) CL command can:

- Remove all deleted records from the file. Because COBOL initializes all relative file records to deleted records, any record that has not been explicitly written will be removed from the file. The relative record numbers of all records after the first deleted record in the file will change.
- Change the relative record numbers if the file has a key and the arrival sequence is changed to match a key sequence (with the KEYFILE parameter).

In addition, a Change Physical File (CHGPF) CL command bearing the REUSEDLT option can change the order of retrieved or written records when the file is operated on sequentially, because it allows the reuse of deleted records.

**COBOL Sequential Files**

A COBOL sequential file is a file in which records are processed in the order in which they were placed in the file, that is, in arrival sequence. For example, the tenth record placed in the file occupies the tenth record position and is the tenth record to be processed. To process a file as a sequential file, you must specify ORGANIZATION IS SEQUENTIAL in the SELECT clause, or omit the ORGANIZATION clause. A sequential file can only be accessed sequentially.

To write standard ANSI X3.23-1985 COBOL programs that access a sequential file, you must create the file with certain characteristics. The following table lists these characteristics and what controls them.
To preserve the sequence of records in a file that you open in I/O (update) mode, do not change the file so that you can reuse the records in it. That is, do not use a Change Physical File (CHGPF) CL command bearing the REUSEDLT option.

**Note:** The COBOL/400 compiler does not check that the device associated with the external file is of the type specified in the device portion of assignment-name. The device specified in the assignment-name must match the actual device to which the file is assigned. See the “ASSIGN Clause” section of the COBOL/400 Reference for more information.

### COBOL File Organization and AS/400 File Access Path Considerations

A file with a keyed sequence access path can be processed in COBOL as a file with INDEXED, RELATIVE, or SEQUENTIAL organization.

For a keyed sequence file to be processed as a relative file in COBOL, it must be a physical file, or a logical file whose members are based on one physical file member. For a keyed sequence file to be processed as a sequential file in COBOL, it must be a physical file, or a logical file that is based on one physical file member and that does not contain select/omit logic.

A file with an arrival sequence access path can be processed in COBOL as a file with RELATIVE or SEQUENTIAL organization. The file must be a physical file or a logical file where each member of the logical file is based on only one physical file member.

When sequential access is specified for a logical file, records in the file are accessed through the access path created with create file options.
File Processing Methods

Figure 82 on page 252 shows the valid processing methods and expected operation for combinations of organization, access mode, open state, I/O verb, and I/O verb modifiers.

All physical database files that are opened for OUTPUT are cleared. Database files with RELATIVE organization, and with dynamic or random access mode, are also initialized with deleted records.

New relative files opened for OUTPUT in sequential access mode are treated differently. Table 4 summarizes conditions affecting them.

<table>
<thead>
<tr>
<th>File Access and CL Specifications</th>
<th>Conditions at Opening Time</th>
<th>Conditions at Closing Time</th>
<th>File Boundary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential *INZDLT</td>
<td>Records not written are initialized</td>
<td>All increments</td>
<td></td>
</tr>
<tr>
<td>Sequential *INZDLT *NOMAX size</td>
<td>CLOSE succeeds File status is 0Q</td>
<td>Up to boundary of records written</td>
<td></td>
</tr>
<tr>
<td>Sequential *NOINZDLT</td>
<td>Records are initialized File is open</td>
<td>Up to boundary of records written</td>
<td></td>
</tr>
<tr>
<td>Random or dynamic</td>
<td>OPEN fails File status is 9Q</td>
<td>File is empty</td>
<td></td>
</tr>
<tr>
<td>Random or dynamic *NOMAX size</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To extend a file boundary beyond the current number of records, but remaining within the file size, use the INZPFM command to add deleted records before processing the file. You need to do this if you receive a file status of 0Q, and you still want to add more records to the file.

Any attempt to extend a relative file beyond its current size results in a boundary violation.

To recover from a file status of 9Q, use the CHGPF command as described in the associated run-time message text.

Lengthy delays are normal when there remains an extremely large number of records (over 1 000 000) to be initialized to deleted records when the CLOSE statement runs.

When the first OPEN statement for the file is not OPEN OUTPUT, relative files should be cleared and initialized with deleted records before they are used. See the discussion of the CLRPFM and INZPFM commands in the CL Reference for more information.
The RECORDS parameter of the INZPFM command must specify "DLT." Overrides are applied when the clear and initialize operations are processed by COBOL, but not when they are processed with CL commands.

Lengthy delays in OPEN OUTPUT processing are normal for extremely large relative files (over 1 000 000 records) that you access in dynamic or random mode.

<table>
<thead>
<tr>
<th>ORG</th>
<th>ACC</th>
<th>DEV</th>
<th>OPEN</th>
<th>READ</th>
<th>WRITE</th>
<th>START</th>
<th>REWRITE</th>
<th>DELETE</th>
<th>CLOSE</th>
<th>FORMAT</th>
<th>SELECT</th>
<th>CLAUSE</th>
<th>KEY IS</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S</td>
<td>ANY</td>
<td>ANY</td>
<td>INPUT</td>
<td>OUTPUT</td>
<td>X</td>
<td>X(F1)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>A1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>S</td>
<td>ANY</td>
<td>ANY</td>
<td>INPUT</td>
<td>OUTPUT</td>
<td>X</td>
<td>X(F1)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>B1</td>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>S</td>
<td>D/DB</td>
<td>D/DB</td>
<td>INPUT</td>
<td>OUTPUT</td>
<td>X</td>
<td>X(F1)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>B1</td>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>S</td>
<td>D/DB</td>
<td>D/DB</td>
<td>INPUT</td>
<td>OUTPUT</td>
<td>X</td>
<td>X(F1)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>B1</td>
<td>D1</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>R</td>
<td>D/DB</td>
<td>D/DB</td>
<td>INPUT</td>
<td>OUTPUT</td>
<td>X</td>
<td>X(F1)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>B1</td>
<td>D1</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>D</td>
<td>D/DB</td>
<td>D/DB</td>
<td>INPUT</td>
<td>OUTPUT</td>
<td>X</td>
<td>X(F1)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>B1</td>
<td>D1</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>S</td>
<td>D/DB</td>
<td>D/DB</td>
<td>INPUT</td>
<td>OUTPUT</td>
<td>X</td>
<td>X(G1)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>C1</td>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>R</td>
<td>D/DB</td>
<td>D/DB</td>
<td>INPUT</td>
<td>OUTPUT</td>
<td>X</td>
<td>X(G1)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>E1</td>
<td>E1</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>D</td>
<td>D/DB</td>
<td>D/DB</td>
<td>INPUT</td>
<td>OUTPUT</td>
<td>X</td>
<td>X(G1)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>E1</td>
<td>E1</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>S</td>
<td>W</td>
<td>I-O</td>
<td>X(KI)</td>
<td>X</td>
<td>X</td>
<td>H1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>D</td>
<td>W</td>
<td>I-O</td>
<td>X(KI)</td>
<td>X</td>
<td>X</td>
<td>I1</td>
<td>J1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 82. Processing Methods Summary Chart**

The following paragraphs explain the keys used in Figure 82.

X The combination is allowed.

A1 The FORMAT phrase is required for FORMATFILE files with multiple formats, and is not allowed for all other device files.

B1 The FORMAT phrase is optional for DATABASE files, and not allowed for DISK files. If the FORMAT phrase is not specified, the default format name of the file is used. The default format name of the file is the first format name defined in the file.

The special register, DB-FORMAT-NAME, can be used to retrieve the format name used on the last successful I/O operation.

C1 The SELECT clause KEY phrase is ignored except for the START statement. If the KEY phrase is not specified on the START statement, the RECORD KEY phrase or the RELATIVE KEY phrase in the SELECT clause is used and KEY = is assumed.
D1 The SELECT clause KEY phrase is used except for the START statement. If the KEY phrase is not specified on the START statement, the RECORD KEY phrase in the SELECT clause is used and KEY = is assumed.

NEXT, PRIOR, FIRST, or LAST can be specified only for the READ statement for DATABASE files with DYNAMIC access. If NEXT, PRIOR, FIRST, or LAST is specified, the SELECT clause KEY phrase is ignored.

E1 The SELECT clause RELATIVE KEY phrase is used.

The NEXT phrase can be specified only for the READ statement for a file with DYNAMIC access mode. If NEXT is specified, the SELECT clause KEY phrase is ignored.

The RELATIVE KEY data item is updated with the relative record number for files with sequential access on READ operations.

F1 A physical file opened for output is cleared.

G1 A physical file opened for output is cleared and initialized to deleted records. There are some exceptions depending on the file size and the options specified. For more information, refer to Table 4 on page 251.

H1 The FORMAT phrase is required for the WRITE statement.

I1 The FORMAT phrase is required to distinguish between the subfile records and the subfile control record. The WRITE FORMAT IS control-record-format-name displays the subfile, but a READ FORMAT IS control-record-format-name is required to allow data to be entered and to cause the operator input for the subfile records on the display to be placed in the subfile.

J1 The SELECT clause RELATIVE KEY phrase is used for READ, WRITE, and REWRITE statements that use the SUBFILE phrase, except that the READ SUBFILE NEXT MODIFIED uses the current system relative record number rather than the RELATIVE KEY data item. The RELATIVE KEY data item is updated with the relative record number for subfile records for READ statements with the NEXT MODIFIED clause.

K1 The SUBFILE phrase is required when an I/O operation deals with a particular record rather than an entire file.

Descending File Considerations

Files created with a descending keyed sequence (in DDS) cause the READ statement NEXT, PRIOR, FIRST, and LAST phrases to work in a fashion exactly opposite that of a file with an ascending key sequence. In descending key sequence, the data is arranged in order from the highest value of the key field to the lowest value of the key field.

For example, READ FIRST retrieves the record with the highest key value, and READ LAST retrieves the record with the lowest key value. Files with a descending key sequence also cause the START qualifiers to work in the opposite manner. For example, START GREATER THAN positions the current record pointer to a record with a key less than the current key.
Chapter 11. COBOL/400 Programming Considerations

This chapter describes:

- Issuing a CL command from a COBOL program
- The CORRESPONDING phrase
- The LIKE clause
- Reference modification
- De-editing
- Performance considerations.

Issuing a CL Command from a COBOL Program

You can issue a CL command from a COBOL program through a CALL to QCMDEXC.

In the following example program, the CALL to QCMDEXC (at sequence number 001600) results in the processing of the Add Library List Entry (ADDLIBLE) CL command (at sequence number 001100). The successful completion of the CL command results in the addition of the library, COBOLTEST, to the library list.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID. CMDXMPLE.
000300 ENVIRONMENT DIVISION.
000400 CONFIGURATION SECTION.
000500 SOURCE-COMPUTER. IBM-AS400.
000600 OBJECT-COMPUTER. IBM-AS400.
000700 DATA DIVISION.
000800 WORKING-STORAGE SECTION.
000900 01 PROGRAM-VARIABLES.
001000 05 CL-CMD PIC X(33) VALUE "ADDLIBLE COBOLTEST".
001100 05 PACK-VAL PIC 9(10)V9(5) COMP-3 VALUE 18.
001200 PROCEDURE DIVISION.
001300 MAINLINE.
001400 CALL "QCMDEXC" USING CL-CMD PACK-VAL.
001500 STOP RUN.
```

Note: Do not use the Reclaim Resource (RCLRSC) Command in this situation. It cancels all programs higher in the program stack so that the STOP RUN statement in the program will cause a run-time exception.

For more information about QCMDEXC, see the CL Programmer’s Guide.

End of General-Use Programming Interface
Using the CORRESPONDING Phrase

In the following example program, the ADD CORRESPONDING statement at sequence number 000270 adds GROUP1 ITEM1 to GROUP2 ITEM1, and adds GROUP1 ITEM2 to GROUP2 ITEM2. The MOVE CORRESPONDING statement at sequence number 000290 moves GROUP1 ITEM1, ITEM2, ITEM3, and ITEM4 to GROUP2 ITEM1, ITEM2, ITEM3, and ITEM4.

The MOVE CORRESPONDING statement at sequence number 000300 is not processed because there are no corresponding items to move, and an error message is generated.

Figure 83 on page 257 was produced with the PRTCORR option in effect.
Figure 83. Example of the CORRESPONDING Phrase
LIKE Clause

The LIKE clause allows you to define the PICTURE, USAGE, SIGN, and BLANK WHEN ZERO characteristics of a data name by copying them from a previously defined data name. LIKE can only refer to a data name or index name, and such names must be uniquely qualified if they have been previously defined. It also allows you to change the length of the data name you define.

This clause is particularly helpful because you can use it to define identifiers in the Working-Storage Section of your program that have the same attributes as variables that you define using the COPY statement.

To create data name DEPTH with the same attributes as data name HEIGHT, write:

DEPTH LIKE HEIGHT

To create data name PROVINCE with the same attributes as data name STATE, except 1 byte longer, write:

PROVINCE LIKE STATE (+1)

This example shows how you can create data item WS-KEY3 with the same attributes as data item KEY3 in the Working-Storage Section:

```
5763C81 V3R0M5  AS/400 COBOL Source
STMT SEQNBR -A 1 B...+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S
  001400  FILE SECTION.
  001500  FD FILE1.
  001600  01 FILE1-REC.
  001700  COPY DDS-ALL-FORMATS OF COPYDDS2.
  +000001  05 COPYDDS2-RECORD PIC X(20).
  +000002*  1-O FORMAT; RECORD1 FROM FILE COPYDDS2 OF LIBRARY COPYLIB
  +000003*  +000004* THE KEY DEFINITIONS FOR RECORD FORMAT RECORD1
  +000005* NUMBER NAME RETRIEVAL TYPE
  +000006* 0001 KEY1-DDS ASCENDING
  +000007* KEYNAME ORIGINATES FROM PHYSICAL FILE
  +000008 05 RECORD1 REDEFINES COPYDDS2-RECORD.
  +000009  06 KEYS PIC X(8).
  +000010  06 FILLER REDEFINES KEYS.
  +000011  07 KEY1-DDS PIC X(4).
  +000012  07 FILLER PIC X(4).
  +000013  06 DATA1 PIC X(12).
  001800 WORKING-STORAGE SECTION.
  001900 01 WS-KEY3 LIKE KEY3.
    PICTURE IS X(8)
```

Figure 84. COPY DDS with the LIKE Clause

The LIKE clause cannot be used in conjunction with the REDEFINES, SIGN, USAGE, or PICTURE clauses. If you use any of these clauses with the LIKE clause, a duplication error occurs. Similarly, BLANK WHEN ZERO can only be specified in conjunction with the LIKE clause if the BLANK WHEN ZERO attribute has not been inherited by the LIKE clause.
A valid LIKE clause has the format of one of the following:

- data-name-1 LIKE-clause xxxxx.
- data-name-1 xxxxx LIKE-clause.
- data-name-1 xxxxx LIKE-clause xxxxx.

The xxxxx is one or a combination of the following clauses: JUSTIFIED, SYNCHRONIZED, BLANK WHEN ZERO, VALUE, OCCURS.

The following show what the LIKE clause can do:

```cobol
01 INCOME.
   05 ANNUAL-WAGES PIC 9(6)V9(2) COMP-3.
   01 YTD-WAGES LIKE ANNUAL-WAGES.
      * PICTURE IS 9(6)V9(2)
      * USAGE IS PACKED-DECIMAL

01 RATES.
   05 MONTHLY-RATE PIC 9(3).
66 GROSS-RATE RENAMES MONTHLY-RATE.
   01 NET-RATE LIKE GROSS-RATE.
      * PICTURE IS 9(3)

01 FAMILY-NAME PIC X(20) VALUE "JONES".
   01 GIVEN-NAME LIKE FAMILY-NAME.
      * PICTURE IS X(20)

01 EMPLOYEE-NUMBER PIC X(6).
   01 DEPARTMENT-MEMBERS.
      05 DEPT-EMPLOYEE-NUMBER LIKE EMPLOYEE-NUMBER OCCURS 10 TIMES.
      * PICTURE IS X(6)

Note: DEPARTMENT-MEMBERS in the above example is 60 bytes long.

   05 TENANT-NAME PIC X(20) OCCURS 10 TIMES.
   01 RENEWAL-RECORD.
      05 RENEWAL-MONTH PIC X(3).
      05 RENEWAL-NAME LIKE TENANT-NAME.
      * PICTURE IS X(20)

Note: RENEWAL-RECORD in the above example is only 23 bytes long.
The PICTURE portion of the generated comment is shown in a concise format.

Note: A numeric field with the BLANK WHEN ZERO attribute is considered to be a numeric edited field.

```cobol
01 ORDER-DETAILS.
   05 ORDER-TYPE PIC XX.
   05 ORDER-CODE LIKE ORDER-TYPE.
   * PICTURE IS X(2)

01 FASTENINGS.
   05 NAILS PIC 9V99 BLANK WHEN ZERO.
   05 RIVETS LIKE NAILS.
   * PICTURE IS 9V9(2)
   * BLANK WHEN ZERO

01 MORTGAGE-PAYMENT.
   05 MORTGAGE-TOTAL PIC S999V99 SIGN IS LEADING SEPARATE.
   05 MORTGAGE-INTEREST LIKE MORTGAGE-TOTAL.
   * PICTURE IS S9(3)V9(2)
   * SIGN IS LEADING SEPARATE

01 PROFIT.
   05 GROSS-PROFIT PIC 999(3)PP(5).
   05 NET-PROFIT LIKE GROSS-PROFIT.
   * PICTURE IS 9(5)P(6)
```

You can use an integer to increase or decrease the length of the field. The following example shows how to increase the field length of WEEKLY-AMOUNT:

```cobol
01 WEEKLY-AMOUNT PIC 9(3).
01 ANNUAL-AMOUNT LIKE WEEKLY-AMOUNT (+3).
   * PICTURE IS 9(6)
```

You should also be aware of the following:

- Any field that has attributes of BLANK WHEN ZERO is considered to be an edited field
- If an integer of zero is specified, an informational message is generated.

Only the integer portion of the field length can be increased or decreased. You cannot change the number of decimal places in a data item.

The default attributes, SIGN IS TRAILING and USAGE IS DISPLAY, are never printed as comments following a LIKE operation.

When you use the LIKE clause, the normal data name qualification rules apply to the parent data name; however, the referenced data name must be uniquely qualified if it has previously been defined more than once. For example:
01 COMBINATIONS.
  05 PHENOTYPE   PIC X(2).
  05 GENOTYPE LIKE PHENOTYPE.
  * PICTURE IS X(2)
 01 PHENOTYPE-TRAITS.
    05 PHENOTYPE   PIC X(3).
    05 PHENO-GROUP LIKE PHENOTYPE OF COMBINATIONS.
    * PICTURE IS X(2)

If you do not uniquely qualify the parent data name, the compiler assigns it a picture clause of X(2), and you receive an error message.

The use of the LIKE clause can sometimes result in group items that are not valid. For example, if you define a COMP-4 group item and then use the LIKE clause to define a COMP-3 item that is subordinate to it, an error will result.

The following example is valid:

```
77 SWITCHES-IN-STOCK PIC S99.
  01 PARTS-ON-ORDER SIGN IS LEADING SEPARATE.
    05 SWITCHES-ON-ORDER LIKE SWITCHES-IN-STOCK.
    * PICTURE IS S9(2)
```

**Note:** SWITCHES-ON-ORDER has the same SIGN attribute (SIGN IS TRAILING) as SWITCHES-IN-STOCK.

In the case of B LIKE A where A is a group item, B cannot be subordinate to A. In all other cases, B will be defined as an alphanumeric item with a length in bytes equal to the length of group A.

```
01 GARAGE-1.
   05 STD-PARKING-1 PIC 9(3).
01 GARAGE-2.
   05 STD-PARKING-2 PIC 9(3) COMP-3.
 77 VACANCIES-1 LIKE GARAGE-1.
   * PICTURE IS X(3)
 77 VACANCIES-2 LIKE GARAGE-2.
   * PICTURE IS X(2)
```

STD-PARKING-1 is a zoned numeric field, so VACANCIES-1 requires 3 bytes of storage. STD-PARKING-2 is a packed numeric field, so VACANCIES-2 requires only 2 bytes of storage.

You can use the LIKE clause with the USAGE IS POINTER clause:

```
01 CUSTOMER-RECORD.
   05 CUST-NAME PIC X(16).
   05 CUST-ADDR-POINTER POINTER.
   05 CUST-STATS-POINTER LIKE CUST-ADDR-POINTER.
   * USAGE IS POINTER
   05 CUST-NUMBER PIC S9(8).
```

**Note:** You cannot use the LIKE clause to change the length of a pointer.

For additional information about the LIKE clause, see the *COBOL/400 Reference.*
Reference Modification

Reference modification allows you to reference substrings of a data item. You simply specify the position within the data item at which you want the substring to start, and the length of the substring. The length is optional: if you omit it, it automatically extends to the end of the data item.

You can write both the starting position and the length value as integer literals, data items, or arithmetic expressions.

The starting position must be at least 1, and cannot be greater than the length of the referenced data item. The length must be at least 1.

The result of adding the starting position to the length specification, then subtracting 1, must fall between 1 and the total length of the referenced data item, inclusive. When the length value is greater than the total length of the data item, an error results.

For additional information on reference modification, see the COBOL/400 Reference.

The *RANGE generation option produces code to detect out-of-range reference modification conditions, and to flag violations with a run-time message.

Suppose you want to retrieve the current time from the system, and display its value in an expanded format. You can retrieve it with the ACCEPT statement, which returns the hours, minutes, seconds, and hundredths of seconds in the format:

```
HHMMSSss
```

However, you may want to view the current time in the format:

```
HH:MM:SS
```

Without reference modification, you must define the following data items:

```
01 TIME-GROUP.
   05 INTERESTING-FIELDS.
      10 HOURS PIC XX.
      10 MINUTES PIC XX.
      10 SECONDS PIC XX.
   05 UNINTERESTING-FIELDS.
      10 HUNDREDTHS-OF-SECONDS PIC XX.
   01 EXPANDED-TIME-GROUP.
      05 INTERESTING-FIELDS.
         10 HOURS PIC XX.
         10 FILLER PIC X VALUE ":".
         10 MINUTES PIC XX.
         10 FILLER PIC X VALUE ":".
         10 SECONDS PIC XX.
```

The following code would retrieve the time value, convert it to its expanded format, and display the new value:
ACCEPT TIME-GROUP FROM TIME
MOVE CORRESPONDING
INTERESTING-FIELDS OF TIME-GROUP TO
INTERESTING-FIELDS OF EXPANDED-TIME-GROUP
DISPLAY "CURRENT TIME IS: " EXPANDED-TIME-GROUP

With reference modification, you do not need to provide names for the subfields that describe the time elements. The only data definition you must have is:

01 REFMOD-TIME-ITEM PIC X(8).

The code to retrieve and expand the time value appears as follows:

ACCEPT REFMOD-TIME-ITEM FROM TIME
DISPLAY "CURRENT TIME IS: 
REFMOD-TIME-ITEM (1:2) :
REFMOD-TIME-ITEM (3:2) :
REFMOD-TIME-ITEM (5:2)

The following example shows a reference beginning at character position 1, for a length of 2, thus retrieving the portion of the time value that corresponds to the number of hours:

REFMOD-TIME-ITEM (1:2)

The following example shows a reference beginning at character position 3, for a length of 2, thus retrieving the portion of the time value that corresponds to the number of minutes:

REFMOD-TIME-ITEM (3:2)

The following example shows a reference beginning at character position 5, for a length of 2, thus retrieving the portion of the time value that corresponds to the number of seconds:

REFMOD-TIME-ITEM (5:2)

Reference Modification with Variable-length Tables

Suppose you are using variable-length tables to contain names:

01 NAME-GROUP.
  05 NAME-LENGTH PIC 99.
  05 NAME-PORTION.
    10 FILLER OCCURS 1 TO 17 TIMES DEPENDING ON NAME-LENGTH.

01 NEW-NAME-GROUP.
  05 NEW-NAME-LENGTH PIC 99.
  05 NEW-NAME-PORTION.
    10 FILLER OCCURS 1 TO 17 TIMES DEPENDING ON NEW-NAME-LENGTH.

The OCCURS DEPENDING ON object of the NAME-PORTION table is set to 8 so that only the first eight occurrences of the table are referenced, even though the entire 17 bytes of NAME-PORTION are filled in.
Suppose you want to change the value in the item NAME-PORTION without changing the portion of the item that is defined beyond the currently defined length. You might try coding:

MOVE NEW-NAME-GROUP TO NAME-GROUP

in which the contents of NEW-NAME-GROUP are:

\[ \begin{array}{c}
0 B L A Y N E \\
T H O M A S J
\end{array} \]

According to the rules for the MOVE statement, the entire contents of the receiving field NAME-GROUP would be replaced. This problem can be avoided by using reference modification in the MOVE statement:

MOVE NEW-NAME-GROUP TO NAME-GROUP ( 1 : LENGTH OF NAME-GROUP )

By specifying the reference modification with the LENGTH OF special register, the length of NAME-GROUP is now determined by the value in the NAME-LENGTH variable.

The new value of NAME-GROUP will be:

\[ \begin{array}{c}
0 B L A Y N E \\
T H O M A S J
\end{array} \]

Reference Modification Using Data Names

So far, all of the reference modification examples have shown simple numeric literals as the reference modification starting position and length values. These values can also be data items or arithmetic expressions.

Suppose a field contains some right-justified characters, and you want to move them to another field, but left-justified instead of right. Using reference modification and an INSPECT statement, you can do it.

The program would have the following data:

```
01 LEFTY                             PIC X(30).
01 RIGHTY                           PIC X(30)      JUSTIFIED RIGHT.
01 I                                PIC 9(9)       USAGE BINARY.
```

The program then counts the number of leading spaces, and, using arithmetic expressions in a reference modification expression, moves the right-justified characters into another (left-justified) field:

```
MOVE SPACES TO LEFTY
MOVE ZERO TO I
INSPECT RIGHTY
   TALLYING I FOR LEADING SPACE
IF I IS LESS THAN 30 THEN
   MOVE RIGHTY ( I + 1 : 30 - I ) TO LEFTY
END-IF
```
The MOVE statement transfers characters from RIGHTY, beginning at the position computed in I + 1, for a length that is computed in 30 – I, into the field LEFTY.

Reference Modification with Subscripting

define a table like this:

```
01 ANY-TABLE.
  05 TABLE-ELEMENT PIC X(10) OCCURS 3 TIMES VALUE "ABCDEFGHIJ".
```

You can change both the third and fourth bytes of the first element of TABLE-ELEMENT to the value “??” with the following MOVE statement:

```
MOVE "??" TO TABLE-ELEMENT ( 1 ) ( 3 : 2 )
```

This statement will move the value “??” into table element number 1, beginning at character position 3, for a length of 2.

ANY-TABLE would look like this before the change:

```
+----------------+
| ABCDEFGHIJ    |
| ABCDEFGHIJ    |
| ABCDEFGHIJ    |
+----------------+
```

It would look like this after the change:

```
+----------------+
| AB??EFGHIJ     |
| ABCDEFGHIJ     |
| ABCDEFGHIJ     |
+----------------+
```

De-editing

**De-editing** allows you to move a numeric-edited data item into a numeric or numeric-edited receiving data item. The compiler accomplishes this by first establishing the unedited value of the numeric-edited item. It then moves the unedited value to the receiver.

De-editing can occur in operations such as MOVE and INITIALIZE. A VALUE clause does not de-edit.

Note that unedited numeric values can involve signs.

Suppose that you use a character field to contain a numeric value that displays on the terminal, and also to contain a value that the computer operator supplies. Suppose that this field has the following definition:

- One character position for a sign (to contain a space if the numeric field is positive or zero, or a minus sign if the numeric field is negative);
- Six digit positions, in which leading zeros are represented by spaces;
• A decimal point;
• Two decimal-digit positions.

The data item that you use to define this field would look like this:

```cobol
01 NUM-EDIT  PIC Z(6).9(2)  USAGE IS DISPLAY.
```

You could initialize this field using this statement:

```
MOVE ZEROS TO NUM-EDIT
```

and when it displays on the terminal, it would contain the value `bbbbb00.00`.

Later, the computer operator might use this field for data entry. If the operator puts `bbbb123.45` into the field, you can obtain the numeric value of the field by moving it into a data item defined as:

```cobol
01 NUMERIC-ITEM  PIC S9(6)V9(2)  USAGE IS PACKED-DECIMAL.
```

This statement:

```
MOVE NUM-EDIT TO NUMERIC-ITEM
```

does de-editing to take place, whereby the numeric item receives the numeric value of the numeric-edited field `NUM-EDIT`. As a result, the numeric item contains the value `+123.45`.

### De-editing Examples

Table 5 and Table 6 show examples of COBOL/400 de-editing.

#### Table 5. Moving Numeric-edited Items into Numeric Receivers

<table>
<thead>
<tr>
<th>Source Picture</th>
<th>Source Value</th>
<th>Receiving Picture</th>
<th>Receiving Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+++;+++;+++</td>
<td><code>b+123.45</code></td>
<td>$S9(5)V9(5) USAGE IS DISPLAY</td>
<td>+123.45</td>
</tr>
<tr>
<td>$+++;+++;+++</td>
<td><code>b–1,234.56</code></td>
<td>$S9(5)V9(5) USAGE IS BINARY</td>
<td>–1234.56</td>
</tr>
<tr>
<td>******.999+</td>
<td>**123.450−</td>
<td>$S9(5)V9(5) USAGE IS PACKED-DECIMAL</td>
<td>–123.45</td>
</tr>
</tbody>
</table>

#### Table 6. Moving Numeric-edited Items into Numeric-edited Receivers

<table>
<thead>
<tr>
<th>Source Picture</th>
<th>Source Value</th>
<th>Receiving Picture</th>
<th>Receiving Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+++;+++;+++</td>
<td><code>b+123.45</code></td>
<td>$$$$$,$$$.$$CR</td>
<td><code>bbbb$123.45bb</code></td>
</tr>
<tr>
<td>$+++;+++;+++</td>
<td><code>b–1,234.56</code></td>
<td>------,------.99</td>
<td><code>bb–1234.56</code></td>
</tr>
<tr>
<td>******.999+</td>
<td>**123.450−</td>
<td>ZZZBZZZBVZZZ</td>
<td><code>bbbb123b450</code></td>
</tr>
<tr>
<td>ZZZ9999CR</td>
<td><code>b+12345bb</code></td>
<td>$+++9999</td>
<td><code>bb+12345</code></td>
</tr>
<tr>
<td>ZZZ9999CR</td>
<td><code>b+12345CR</code></td>
<td>999999.99−</td>
<td>012345.00−</td>
</tr>
</tbody>
</table>
Handling Data Errors

The compiler provides some run-time error checking for move operations that involve de-editing.

The compiler does not perform this checking for source values of zero, and it ignores simple insertion characters (such as / B 0 , ).

Sign Test

The compiler validates signs in numeric-edited source items according to the following rules.

<table>
<thead>
<tr>
<th>PICTURE Definition</th>
<th>Allowable Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed +</td>
<td>+ or –</td>
</tr>
<tr>
<td>Fixed –</td>
<td>b or –</td>
</tr>
<tr>
<td>CR</td>
<td>bb or CR</td>
</tr>
<tr>
<td>DB</td>
<td>bb or DB</td>
</tr>
</tbody>
</table>

If these rules are disobeyed, a sign error occurs, and the program stops.

Float Test

If the source has a string of floating characters, this test verifies the correctness of leading floating characters in the data field.

The rules for the float test are:

- If the source PICTURE clause contains floating $ symbols, the first non-blank character in the relevant portion of the source field (positions 2 through 7 in the example) must be a $, and its location must be correct according to the rules for PICTURE clause editing. (See the COBOL/400 Reference for more information about these rules.)

For example:

```
Location of a Leading Floating Character

01 A PIC +$$B,$$$.
   .
   /* Note that "b" represents one space */
   /* PIC String: +$$B,$$$ */
   /* Position indexes: 12345678 */
   MOVE 1 TO A. /* A = "b$bb$1" */
   MOVE 12 TO A. /* A = "b$bb$12" */
   MOVE 123 TO A. /* A = "b$bb$123" */
   MOVE 1234 TO A. /* A = "b$1b,234" */
```

In this example, the $ must be located at position 2, 5, 6, or 7.

- If the source PICTURE clause contains floating + symbols, the first non-blank character in the relevant portion of the source field must be + or –, and its location must be correct according to the rules for PICTURE clause editing.
• If the source PICTURE clause contains floating – symbols, the relevant portion of the source field must start with:
  – One or more contiguous spaces, the last of which must be correctly located according to the rules for PICTURE clause editing
  – One or more contiguous spaces, with a – immediately following it. The location of the – must be correct according to the rules for PICTURE clause editing.
  – A –.

If these rules are disobeyed, a float error occurs, and the program stops.

---

**Performance Considerations**

**PICTURE Clauses for Numeric Items**

Because hardware instructions use signs, you can improve performance by including an S in a picture clause whenever possible.

You can also improve performance by specifying odd numbers of numeric character positions in the picture clauses for COMP-3 (packed decimal) items. Internally, the rightmost byte of a packed decimal item contains a digit and a sign, and any other bytes contain two digits. If you use the more efficient configuration, the compiler does not need to supply the missing digit.

**Eight-Byte Binary Items**

Avoid using 8-byte binary items. You can specify these items for convenience, but the compiler must make conversions in order to use them.

**Segmentation**

Use of segmentation increases the compile and run times of the COBOL program. The segmentation feature is provided only for compatibility with other systems. You do not have to be concerned with storage management when using COBOL/400 programs.

**Calling a COBOL Program from a Non-COBOL Program**

Repeated calls of a COBOL program from a non-COBOL program can result in a marked decrease in compiler performance due to the fact that exiting from the main COBOL program (the program that initiated the COBOL run unit) causes the program to be deactivated.
A new function, MGTFUNC has been added to the COBOL run-time routine, QRLMAIN to prevent this deactivation by causing the main COBOL program to be treated as a subprogram. Because this fix depends on the size of MGT, it is recommended that the run-time routine, QLRMAIN be called from the main COBOL program with MGTFUNC = 9, as shown in the following example:

```cobol
01 mgtstruc.
   03 FILLER PIC X(277).
   03 mgtfunc pic 9(2) comp-4 value 9.
   77 TEST-VAR PIC X(10) value spaces.

   if test-var = spaces then
     display 'spaces'
     move 'faked' to test-var
     call 'QLRMAIN' using mgtstruc
   else
     display 'not spaces' test-var.
```

Notes:
1. The 01 mgtstruc must be on a 16 byte boundary. If a boundary error occurs, add 77 aa PIC X. in front of the 01.
2. Because the call to QLRMAIN changes the main COBOL program to a subprogram, you should use the EXIT PROGRAM command and not STOP RUN, which may cause errors.
3. RCLRSC will deactivate the main program (now a subprogram)

Debugging

COBOL source language debugging is provided to help the COBOL programmer debug a program that is not functioning as expected. Use of this facility increases the compile and run times of a COBOL program.

*NORANGE Option

This GENOPT parameter option of the CRTCBBLPGM command removes the run-time checks for subscript and reference modification ranges.

This option can improve performance when:
- You make frequent references to tables, and the subscripts always reference elements that are in the tables
- You use reference modification often.

Note: The *RANGE option generates code for checking subscript ranges. For example, it ensures that you are not attempting to access the 21st element of a 20-element array.

The *NORANGE option does not generate code to check subscript or reference modification ranges.

These options do not eliminate the zero subscript checking performed by the operating system. If zero subscripts occur, the operating system will not permit their use and issues message MCH0603.
*DUPKEYCHK Option

This GENOPT parameter option of the CRTCBLPGM command indicates that duplicate key checking for INDEXED files will be performed. Using DUPKEYCHK while reading INDEXED files can adversely affect performance.

Relative Files

You can experience lengthy delays if you open or close relative files in which very large volumes of records are being initialized to deleted records.

See Table 4 on page 251 for more information.

Indicators

If you use indicators in a separate indicator area (INDARA keyword specified in DDS) instead of in the record area, the use of the OCCURS clause to specify a table with up to 99 indicators can improve performance. See Figure 60 on page 155 for more information.

Commitment Control

Generally, the use of commitment control increases the run time of a COBOL program. In addition, the record locking that results from the use of commitment control by a job may cause delays for other users attempting to access the same file.

Reading without Record Locks

To avoid unnecessary record locks, you can include the NO LOCK phrase in your READ statement. For more information about this phrase, refer to the section on the READ statement in the COBOL/400 Reference.

Initializing Variables

You can reduce program run time by choosing not to initialize program variables that have no value clauses associated with them. You can specify no initialization by specifying *NOSTDINZ for the GENOPT parameter of the CRTCBLPGM command, or by specifying NOSTDINZ in the PROCESS statement. The compiler then initializes only those variables that have value clauses declared. An additional benefit to this option is that you can also compile larger programs with a greater number of variables.

If you specify *NOSTDINZ, you must ensure that all data items contain valid data before you attempt to manipulate the items. If an item does not contain valid data, decimal data errors can occur.

Blocking Records

You can use record blocking to improve your run-time performance. The key benefits for blocking are realized when you read multiple records sequentially, such as a random read followed by sequential reads.

For information on blocking, refer to “Unblocking Input Records and Blocking Output Records” on page 102.
Program Loops

When a program repeatedly processes the same series of instructions, and it is apparent that this will continue indefinitely, the program is in a loop. To identify loops, you can use information known about the program itself, as follows:

- Time: If the actual run time is substantially exceeding the expected run time, the program could be in a loop.
- I/O operations: If no input/output operations are taking place and I/O is expected to be occurring repeatedly, the program is probably in a loop.

Tracing a Loop in a Program

Frequently, a loop encompasses many instructions in a program. In this case, you can use the COBOL debugging features as described in Chapter 5, “Debugging Your Program” on page 55.

Errors That Can Cause a Loop

A PERFORM statement with an UNTIL clause can cause a loop when the condition specified in the UNTIL clause cannot be met. For example:

PERFORM ... UNTIL COUNTR LESS THAN ZERO

where COUNTR is an unsigned numeric item.

A GO TO statement that refers to a previous procedure-name can cause a loop when no conditional statement exists to prevent the GO TO statement from being processed again. For example:

PARA-1.
  MOVE ...
  MOVE ...
  MOVE ...
PARA-2.
  MOVE ...
  GO TO PARA-1.

A possible variation of this case occurs when a conditional statement exists, but the condition cannot be met or the statement does not branch (through a GO TO statement) to a paragraph outside the range of the loop.
Chapter 12. Communicating Between Programs

Sometimes an application is simple enough to be coded as a single, self-sufficient program. In many cases, however, an application's solution will consist of several, separately compiled programs used together.

The AS/400 system provides communication between COBOL programs, and between COBOL and non-COBOL programs.

A COBOL run unit is a set of one or more programs that function as a unit at run time to provide a problem solution. A COBOL run unit starts with the first COBOL program in the program stack, and includes all programs (of any type) that are below it. A program stack is a list of programs linked together as a result of programs calling other programs, or implicitly from some other event within the same job.

When a run unit consists of several, separately compiled programs that call each other, the programs must be able to communicate with each other. They need to transfer control and usually need to have access to common data. This chapter describes the methods that accomplish this interprogram communication between separately compiled programs.

Transferring Control to Another Program

In the Procedure Division, a program can call another program (generally called a subprogram in COBOL terms), and this called program may itself call another program. The program that calls another program is referred to as the calling program, and the program it calls is referred to as the called program.

The called COBOL program starts running at the top of the Procedure Division.

When the called program processing is completed, the program can either transfer control back to the calling program or end the run unit.

A called program must not directly or indirectly execute its caller (such as program X calling program Y; program Y calling program Z; and program Z then calling program X). This is called a recursive call. COBOL/400 allows recursion in both main programs and subprograms. However, if you want your programs to conform to SAA standards, do not use recursive calls.

Main Programs and Subprograms

The first COBOL program to be executed begins the COBOL run unit, and is the main program. No specific source statements or options identify a COBOL program to be a main program or a subprogram. A subprogram is a program in the run unit below the main program in the program stack. For more information about program stacks and other terms concerning interprogram communication, see the CL Programmer’s Guide.
Returning Control from a Called Program

It is important to know if a COBOL program is a main program or a subprogram to determine how control is returned from a called program when an error occurs, or a program ends.

You can issue a STOP RUN, EXIT PROGRAM, or GOBACK statement to return control from a called program.

If execution ends in the main program, either STOP RUN or GOBACK is used. These statements end the run unit, and control is returned to the caller of the main program.

If execution ends in a subprogram, the subprogram may end with an EXIT PROGRAM, a GOBACK, or a STOP RUN statement. If the subprogram ends with an EXIT PROGRAM or a GOBACK statement, control returns to its immediate caller without ending the run unit. An implicit EXIT PROGRAM statement is generated if there is no next executable statement in a called program. If it ends with a STOP RUN statement, the effect is the same as it is in a main program: all COBOL programs in the run unit are terminated, and control returns to the caller of the main program.

A subprogram is left in its last-used state when it terminates with EXIT PROGRAM or GOBACK. The next time it is called in the run unit, its internal values will be as they were left, except that return values for PERFORM statements will be reset to their initial values. In contrast, a main program is initialized each time it is called.

The following examples illustrate the use of the EXIT PROGRAM and STOP RUN statements in different parts of a run unit.

- The example in Figure 85 on page 275 shows a single run unit.
- The example in Figure 86 on page 276 shows multiple run units that run consecutively
- The example in Figure 87 on page 277 shows a run unit with a shared program that is both a subprogram and a main program.
- The example in Figure 88 on page 278 shows multiple run units that run concurrently.

Note: You can substitute a GOBACK statement for an EXIT PROGRAM statement that appears in a subprogram, or a STOP RUN statement that appears in a main program.
No operation is processed because the statement is processed in a main program. Processing continues with the next statement in PGMA.

Control returns to the caller of the program that processes the EXIT PROGRAM statement.

Run unit A ends. For all programs in the run unit, open files are closed. Storage is freed for all programs in the run unit. Control returns to the program that is at call level n-1. If n=1, the following considerations apply:

- Run unit A operates as a job step. See the CL Programmer’s Guide for more information.
- For batch jobs, the STOP RUN statement ends the job. For interactive jobs, control returns to the system and the system ends the job step.
Figure 86. Example of Multiple Run Units That Run Consecutively

1. No operation is processed because the statement is processed in a main program. Processing continues with the next statement in the main program.

2. Control returns to the caller of the program that processes the EXIT PROGRAM statement.

3. Run unit B ends. All open files in run unit B are closed. Storage is freed for all programs in run unit B. Control returns to the caller of the main program for the run unit (PGMA).

4. Run unit C ends. All open files in run unit C are closed. Storage is freed for all programs in run unit C. Control returns to the caller of the main program for the run unit (PGMA).
Figure 87. Example of a Run Unit with a Shared Program that is Both a Subprogram and a Main Program

1. No operation is processed because the statement is processed in a main program. Processing continues with the next statement in the main program.

2. Control returns to the caller of the program that processes the EXIT PROGRAM statement.

3. Run unit B ends. All open files in run unit B are closed. Storage is freed for all programs in run unit B. Control returns to the caller of the main program for the run unit (PGMA).
4 Run unit E ends. All open files in run unit E are closed. Storage is freed for PGME. Control returns to the caller of the main program for the run unit (PGMC).

5 Run unit F ends. All open files in run unit F are closed. Storage is freed for PGMF. Control returns to the caller of the main program for the run unit (PGMC).

Figure 88. Example of Multiple Run Units That Run Concurrently

1 No operation is processed because the statement is processed in a main program. Processing continues with the next statement in the main program.

2 Control returns to the caller of the program that processes the EXIT PROGRAM statement.
Run unit B can only end after run unit E completes a STOP RUN. When run unit B ends, all open files in run unit B are closed. Storage is freed for all programs in run unit B, and control returns to the caller of the main program (PGMA).

Run unit E ends. All open files in run unit E are closed. Storage is freed for all programs in run unit E. Control returns to PGMD in run unit B.

Concurrent run units are achieved by using the QLRCHGCM API. Refer to the System Programmer’s Interface Reference for more information on this API.

Initialization of Storage

The first time a COBOL program in a run unit is called, its storage is initialized. Storage is initialized again under the following conditions:

- The run unit is terminated, then reinitiated.
- The program is canceled (using the CANCEL statement for COBOL, the FREE operation for the RPG/400* programming language, or the Reclaim Resource (RCLRSC) command), and then called again.

If a non-COBOL program is named in a CANCEL statement, its name must conform to the rules for formation of a COBOL program name.

Calling Another Program

You will often want your COBOL programs to communicate with other COBOL and non-COBOL programs.

Passing Data Using BY REFERENCE or BY CONTENT

BY REFERENCE means that the subprogram is referring to and processing the data items in the calling program's storage, rather than working on a copy of the data.

BY CONTENT means that the calling program is passing only the contents of the literal or identifier. With a CALL . . . BY CONTENT, the called program cannot change the value of the literal or identifier in the calling program, even if it modifies the variable in which it received the literal or identifier.

Whether you pass data items BY REFERENCE or BY CONTENT depends on what you want your program to do with the data:

- If you want the definition of the argument of the CALL statement in the calling program and the definition of the parameter in the called program to share the same memory, specify:
  
  CALL . . . BY REFERENCE identifier.

  Any changes made by the subprogram to the parameter affect the argument in the calling program.

  An identifier in the USING phrase of the CALL . . . BY REFERENCE statement may be a file-name, in addition to a data-name.

  File-names as CALL operands are allowed by the compiler as an extension.
If you want to pass the address of a record area to a called program, specify:

```
CALL ... BY REFERENCE ADDRESS OF record-name.
```

The subprogram receives the ADDRESS OF special register for the record-name you specify.

You must define the record name as a level-01 or level-77 item in the Linkage Section of the called and calling programs. A separate ADDRESS OF special register is provided for each record in the Linkage Section.

If you do not want the definition of the argument of the CALL statement in the calling program and the definition of the parameter in the called subprogram to share the same memory, specify:

```
CALL ... BY CONTENT identifier.
```

If you want to pass a literal value to a called program, specify:

```
CALL ... BY CONTENT literal.
```

The called program cannot change the value of the literal. The literal cannot be numeric.

If you want to pass the length of a data item, specify:

```
CALL ... BY CONTENT LENGTH OF identifier.
```

The calling program passes the length of identifier from its LENGTH OF special register. When literals are passed BY CONTENT, the called program cannot change their values.

If you want to pass both a data item and its length to a subprogram, specify a combination of BY REFERENCE and BY CONTENT. For example:

```
CALL 'ERRPROC' USING BY REFERENCE A
    BY CONTENT LENGTH OF A.
```

Data items in a calling program can be described in the Linkage Section of all the programs it calls directly or indirectly. In this case, storage for these items is allocated in the highest calling program. That is, program A calls program B, which calls program C. Data items in program A can be described in the Linkage Sections of programs B and C, so that one set of data can be made available to all three programs.

**Describing Arguments in the Calling Program**

In the calling program, the arguments are described in the Data Division in the same manner as other data items in the Data Division. Unless they are in the Linkage Section, storage is allocated for these items in the calling program. If you reference data in a file, the file must be open when the data is referenced. Code the USING clause of the CALL statement to pass the arguments.

**Describing Parameters in the Called Program**

In the called program, parameters are described in the Linkage Section. Code the USING clause after the PROCEDURE-DIVISION header to receive the parameters.
In the Linkage Section

You must know what is being passed from the calling program and set up the Linkage Section in the called program to accept it. To the called program, it doesn't matter which clause of the CALL statement you use to pass the data (BY REFERENCE or BY CONTENT). In either case, the called program must describe the data it is receiving. It does this in the Linkage Section.

The number of data-names in the identifier list of a called program must not be greater than the number of data-names in the identifier list of the calling program. There is a one-to-one positional correspondence; that is, the first identifier of the calling program is passed to the first identifier of the called program, and so forth. The compiler makes no attempt to match arguments and parameters.

Grouping Data to be Passed

Consider grouping all the data items you want to pass between programs and putting them under one level-01 item. If you do this, you can pass a single level-01 record between programs. For an example of this method, see Figure 89.

To make the possibility of mismatched records even smaller, put the level-01 record in a copy member, and copy it in both programs. (That is, copy it in the Working-Storage Section of the calling program and in the Linkage Section of the called program.)

<table>
<thead>
<tr>
<th>Calling Program Description</th>
<th>Called Program Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORKING─STORAGE SECTION.</td>
<td>LINKAGE SECTION.</td>
</tr>
<tr>
<td>01 PARAM─LIST.</td>
<td>01 USING─LIST.</td>
</tr>
<tr>
<td>05 PARTCODE PIC A.</td>
<td>10 PART─ID PIC X(5).</td>
</tr>
<tr>
<td>05 PARTNO PIC X(4).</td>
<td>10 SALES PIC 9(5).</td>
</tr>
<tr>
<td>05 U─SALES PIC 9(5).</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>PROCEDURE DIVISION.</td>
<td>PROCEDURE DIVISION</td>
</tr>
<tr>
<td></td>
<td>USING USING─LIST.</td>
</tr>
<tr>
<td>CALL CALLED─PROG</td>
<td></td>
</tr>
<tr>
<td>USING PARAM─LIST.</td>
<td></td>
</tr>
</tbody>
</table>

In the calling program, the code for parts (PARTCODE) and the part number (PARTNO) are referred to separately.

In the called program, the code for parts and the part number are combined into one data item (PART─ID). In the called program, a reference to PART─ID is the only valid reference to them.

Figure 89. Common Data Items in Subprogram Linkage
Call by Identifier

A system pointer that associates an identifier with an object is set the first time you use the identifier in a CALL statement.

--- Important for compatibility! ---

If you carry out a call by an identifier to a program that you subsequently delete or rename, you must use the CANCEL statement to null the system pointer associated with the identifier. This ensures that when you next use the identifier to call your program, the associated system pointer will be set again.

The following example shows how to apply the CANCEL statement to an identifier:

```
MOVE "ABCD" TO IDENT-1.
CALL IDENT-1.
CANCEL IDENT-1.
```

If you apply the CANCEL statement directly to the literal "ABCD", you do not null the system pointer associated with IDENT-1. Instead, you can continue to call program ABCD simply by using IDENT-1 in your CALL statement.

The value of the system pointer also changes if you change the value of the identifier and perform a call using this new value.

Using Pointers in a COBOL/400 Program

You can use a pointer (a data item in which address values can be stored) within a COBOL program when you want to pass and receive addresses of a variably-located data item, and to accomplish limited base addressing.

On the AS/400 system, pointers are 16 bytes long. COBOL pointers are AS/400 space pointers since they point to system space objects. One part of the pointer describes its attributes, such as which AS/400 space object it is pointing to. Another part of the pointer contains the offset into the AS/400 system space object.

To define a COBOL pointer, called a pointer data item, code a USAGE IS POINTER clause on the data item. A pointer data item is a 16-byte elementary item that can be compared for equality, or used to set the value of other pointer items.

A pointer data item can be used only in:

- A SET statement (Format 5 only)
- A relation condition
- The USING phrase of a CALL statement, or the Procedure Division header.
- The operand for the LENGTH OF and ADDRESS OF special registers.

If pointers are used in a relational condition, the only valid operators are equal to, or not equal to.

Because pointer data items are not simply binary numbers on the AS/400 system, manipulating pointers as integers does not work.
Pointer data items are defined explicitly with the USAGE IS POINTER clause, and are implicit when using an ADDRESS OF special register or the ADDRESS OF an item.

If a group item is described with the USAGE IS POINTER clause, the elementary items within the group item are pointer items. The group itself is not a pointer data item, and cannot be used in the syntax where a pointer data item is allowed. The USAGE clause of an elementary item cannot contradict the USAGE clause of a group to which the item belongs.

Pointer data items can be part of a group that is referred to in a MOVE statement or an input/output statement; however, if a pointer data item is part of a group, there is no conversion of pointer values to another form of internal representation when the statement is executed.

**Defining Pointers and Pointer Alignment**

Pointer data items can be defined at any level (except 88) in the File, Working-Storage, or Linkage sections of a program.

When a pointer is referenced on the AS/400 system, it must be on a 16-byte storage boundary. **Pointer alignment** refers to the COBOL/400 compiler's process of positioning pointer items within a group item to offsets that are multiples of 16 bytes from the beginning of the record. If a pointer item is not on a 16-byte boundary, a pointer alignment exception (MCH0602) is sent to the COBOL/400 program. In general, pointer alignment exceptions occur in the Linkage Section, where it is up to the user to align these items.

In the File and Working-Storage sections, the compiler ensures that this exception does not occur by adding implicit FILLER items. Every time an implicit FILLER item is added by the compiler, a warning is issued. In the Linkage Section, no implicit FILLER items are added by the compiler; however, warnings are issued indicating how many bytes of FILLER would have been added had the group item appeared in the File or Working-Storage sections.

You can define a data item as a pointer by specifying the USAGE IS POINTER clause as shown in the following example:

```
WORKING-STORAGE SECTION.
  77 APTR USAGE POINTER.
  01 AB.
    05 BPTR USAGE POINTER.
    05 BVAR PIC S9(3) PACKED-DECIMAL.
  LINKAGE SECTION.
  01 AVAR.
    05 CVAR PIC X(3).
PROCEDURE DIVISION.
  SET APTR TO ADDRESS OF AVAR.
```

*Figure 90. Defining a Pointer Data Item*

In the above example, AVAR is an 01-level data item, so the ADDRESS OF AVAR is the ADDRESS OF special register. Because a special register is an actual storage area, the SET statement moves the contents of ADDRESS OF AVAR into pointer data item APTR.
In the above example, if the SET statement used ADDRESS OF CVAR, no special register exists. Instead, the pointer data item APTR is assigned the calculated address of CVAR.

In File and Working-Storage Sections
In the File and Working-Storage sections, all 01-level items (and some 66 and 77-level items) are placed on 16-byte boundaries.

Within a group structure, pointer data items must also occur on a 16-byte boundary. To ensure this, the COBOL/400 compiler adds FILLER items immediately before the pointer data item. To avoid these FILLER items, you should place pointer data items at the beginning of a group item.

If the pointer data item is part of a table, the first item in the table is placed on a 16-byte boundary. To ensure that all subsequent occurrences of the pointer fall on a 16-byte boundary, a FILLER item is added to the end of the table if necessary.

An example of pointer data item alignment follows:

```cobol
WORKING-STORAGE SECTION.
  77 APTR USAGE POINTER.
  01 AB.
    05 ALPHA-NUM PIC X(10).
    05 BPTR USAGE POINTER.
  01 EF.
    05 ARRAY-1 OCCURS 3 TIMES.
      10 ALPHA-NUM-TWO PIC X(14).
      10 CPTR USAGE POINTER.
      10 ALPHA-NUM-THREE PIC X(5).
```

Figure 91. Aligning Pointer Data Items

In the above example, APTR is a pointer data item. The 77-level item, therefore, is placed on a 16-byte boundary. The group item AB is an 01-level item and is automatically placed on a 16-byte boundary. Within the group item AB, BPTR is not on a 16-byte boundary. To align it properly, the compiler inserts a 6-byte FILLER item after ALPHA-NUM. Finally, CPTR requires a FILLER of 2 bytes to align its first occurrence. Because ALPHA-NUM-THREE is only 5 bytes long, another 11-byte FILLER must be added to the end of ARRAY-1 to align all subsequent occurrences of CPTR.

When a pointer is defined in the File Section, and a file does not have blocking in effect, each 01-level item will be on a 16-byte boundary. If a file has blocking in effect, only the first record of a block is guaranteed to be on a 16-byte boundary. Thus pointer data items should not be defined for files with blocking in effect. For more information on blocking, refer to “Unblocking Input Records and Blocking Output Records” on page 102.

Pointers and the REDEFINES Clause
A pointer data item may be the subject or object of a REDEFINES clause.

When a pointer is the subject of a REDEFINES clause, the object data item must be on a 16-byte boundary.
For example:

```
WORKING-STORAGE SECTION.
  01 AB.
    05 ALPHA-NUM PIC X(16).
    05 APTR REDEFINES ALPHA-NUM USAGE POINTER.
    05 BPTR USAGE POINTER.
    05 CPTR REDEFINES BPTR USAGE POINTER.
```

*Figure 92. REDEFINES and Aligned Pointer Data Items*

In the above example, both APTR and CPTR are pointer data items that redefine 16-byte aligned items. In the following example, the redefined item would result in a severe compiler error:

```
WORKING-STORAGE SECTION.
  01 EF.
    05 ALPHA-NUM PIC X(5).
    05 HI.
      10 ALPHA-NUM-TWO PIC X(11).
      10 APTR USAGE POINTER.
      05 BPTR REDEFINES HI USAGE POINTER.
```

*Figure 93. REDEFINES and Aligned Pointer Data Items - Incorrect Method*

In the above example, APTR is aligned on a 16-byte boundary. That is, the COBOL/400 compiler did not need to add FILLER items to align APTR. The group item HI is not on a 16-byte boundary, and so neither is pointer data item BPTR. Because the COBOL/400 compiler cannot add FILLER items to place BPTR on a 16-byte boundary, a severe error will result. In the following example, similar to the above, the COBOL/400 compiler is able to place the pointer data item on a 16-byte boundary:

```
WORKING-STORAGE SECTION.
  01 EF.
    05 ALPHA-NUM PIC X(5).
    05 HI.
      10 ALPHA-NUM-TWO PIC X(11).
      10 APTR USAGE POINTER.
      10 ALPHA-NUM-THREE PIC X(5).
      05 KL REDEFINES HI.
      10 BPTR USAGE POINTER.
```

*Figure 94. REDEFINES and Unaligned Pointer Data Items - Correct Method*

In the above example, group item KL is not on a 16-byte boundary; however, the compiler adds an 11-byte FILLER before pointer data item BPTR to ensure that it falls on a 16-byte boundary.

---

**Reading and Writing Pointers**

Pointer data items can be defined in the File Section, and can be set and used as can any other Working-Storage pointer data items. There are, however, some restrictions:

- If a file has blocking in effect, only the first record of a block is guaranteed to be on a 16-byte boundary. Thus pointer data items should not be defined for files with blocking in effect.
- A record containing pointers can be written to a file; however, on subsequent reading of that record, the pointer data items equal NULL.

### Initializing Pointers Using the NULL Figurative Constant

The NULL figurative constant represents a value used to indicate that data items defined with USAGE IS POINTER, ADDRESS OF, or the ADDRESS OF special register do not contain a valid address. For example:

```cobol
WORKING-STORAGE SECTION.
  77 APTR USAGE POINTER VALUE NULL.
PROCEDURE DIVISION.
  IF APTR = NULL THEN
    DISPLAY 'APTR IS NULL'
  END-IF.
```

*Figure 95. Using NULL to Initialize a Pointer*

In the above example, pointer APTR is set to NULL in the Working-Storage section. The comparison in the procedure division will be true and the display statement is executed.

On the AS/400 system, the initial value of a pointer data item with or without a VALUE clause of NULL, equals NULL.

### LENGTH OF Special Register

The LENGTH OF special register contains the number of bytes used by an identifier. It returns a value of 16 for a pointer data item.

You can use LENGTH OF in the Procedure Division anywhere a numeric data item having the same definition as the implied definition of the LENGTH OF special register is used; however, LENGTH OF cannot be used as a subscript or a receiving data item. LENGTH OF has the implicit definition:

```
USAGE IS BINARY, PICTURE 9(9)
```

The following example shows how you can use LENGTH OF with pointers:

```cobol
WORKING-STORAGE SECTION.
  77 APTR USAGE POINTER.
  01 AB.
    05 BPTR USAGE POINTER.
    05 BVAR PIC S9(3) PACKED-DECIMAL.
    05 CVAR PIC S9(3) PACKED-DECIMAL.
PROCEDURE DIVISION.
  MOVE LENGTH OF AB TO BVAR.
  MOVE LENGTH OF BPTR TO CVAR.
```

*Figure 96. Using LENGTH OF with Pointers*

In the above example, the length of group item AB is moved to variable BVAR. BVAR has a value of 20 because BPTR is 16 bytes long, and both variables BVAR and CVAR are 2 bytes long. CVAR receives a value of 16.

You can also use the LENGTH OF special register to set up data structures within user spaces, or to increment addresses received from another program. To see an
example of a program that uses the LENGTH OF special register to define data structures within user spaces, refer to Figure 99 on page 291.

Setting the Address of Linkage Items

Generally, when one COBOL program calls another, data passes between the two programs in the following manner: the calling program uses the CALL USING statement to pass operands to the called program, and the called program specifies the USING phrase in the Procedure Division header. There should be a one-to-one mapping between the operands in the USING phrases of each program.

When using the ADDRESS OF special register, you no longer need to ensure a one-to-one mapping between the USING phrases of the two programs. For those data items in the Linkage Section that are not specified in the USING phrase of the Procedure Division header, you can use a SET statement to specify the starting address of the data structure. Once the SET statement is run, the data item is then treated as if it was passed from another program. For an example of a SET statement used in this manner, refer to Figure 100 on page 292. On page 295 illustrates how the SET statement is used to set the starting address of the data structures is-header-record and is-user-space at the beginning of the user space.

Using ADDRESS OF and the ADDRESS OF Special Register

When you specify ADDRESS OF in a COBOL program, the compiler determines whether to use the calculated address of a data item, referred to as ADDRESS OF, or the ADDRESS OF special register. The ADDRESS OF special register is the starting address of the data structure from which all calculated addresses are determined. Because the ADDRESS OF special register is the starting address of a structure, it must be an 01-level or 77-level data item. If you reference modify this data item, it is no longer the starting address of the data structure. It is a calculated address, or ADDRESS OF. If you are taking the ADDRESS OF an elementary item, and the ADDRESS OF the 01-level item has been set to NULL, a pointer exception (MCH3601) results.

You cannot use the calculated ADDRESS OF where an item can be changed. Only the ADDRESS OF special register can be changed. For example, in Figure 100, the SET statement at on page 295 uses the ADDRESS OF special register because it is an 01-level item. At on page 295 ADDRESS OF is used because, although it is an 01-level item, it is reference-modified.

Using Pointers in a MOVE Statement

Elementary pointer data items cannot be moved using the MOVE statement; a SET statement must be used; however, pointer data items are implicitly moved when they are part of a group item.

When compiling a MOVE statement, the COBOL/400 compiler generates code to maintain (a pointer MOVE) or not maintain (a non-pointer MOVE) pointers within a group item.

A pointer MOVE is done when all of the following conditions are met:

1. The source or receiver of a MOVE statement contains a pointer
2. Both of the items are at least 16 bytes long
3. The data items are properly aligned
4. The data items are alphanumeric or group items.

Of the conditions listed above, determining if two data items are properly aligned can be the most difficult.

If the items being moved are 01-level items, or are part of an 01-level item, they must be on the same offset relative to a 16-byte boundary for a pointer MOVE to occur. (A warning is issued if this is not true.) The following example shows three data structures, and the results when a MOVE statement is issued:

WORKING-STORAGE SECTION.
  01 A.
    05 B PIC X(10).
    05 C.
      10 D PIC X(6).
      10 E POINTER.
  01 A2.
    05 B2 PIC X(6).
    05 C2.
      10 D2 PIC X(10).
      10 E2 POINTER.
  01 A3.
    05 B3 PIC X(22).
    05 C3.
      10 D3 PIC X(10).
      10 E3 POINTER.

PROCEDURE DIVISION.
MOVE A to A2. 1
MOVE A to A3. 1
MOVE C to C2. 2
MOVE C2 to C3. 3

1  This results in a pointer move because the offset of each group item to be moved is zero. Pointer integrity is maintained.

2  This results in a non-pointer move, because the offsets do not match. The offset of group item C is 10, and the offset of group item C2 is 6. Pointer integrity is not maintained.

3  This results in a pointer move, because the offset of group item C2 is 6, and the offset of C3 relative to a 16-byte boundary is also 6. (When the offset is greater than 16, the offset relative to a 16-byte boundary is calculated by dividing the offset by 16. The remainder is the relative offset. In this case, the offset was 22, which, when divided by 16, leaves a remainder, or relative offset, of 6.) Pointer integrity is maintained.

If a group item contains a pointer, and the compiler cannot determine the offset relative to a 16-byte boundary, the compiler issues a warning message, and the pointer move is attempted. However, pointer integrity may not be maintained. The compiler cannot determine the offset if the item is defined in the Linkage Section, or if the item is reference-modified with an unknown starting position. You must ensure that pointer alignment is maintained, or MCH0602 may result.
The COBOL/400 compiler places all 01-level items on a 16-byte boundary whether or not they contain pointer data items.

If one of the items in a MOVE statement is an 01-level item with a pointer, and the other a 77-level Working-Storage item, the 77-level Working-Storage item is forced to a 16-byte boundary.

### Using Pointers in a CALL Statement

When a pointer data item is passed in a CALL statement, the item is treated as all other USING items. In other words, a pointer to the pointer data item (or copy of the pointer data item) is passed to the called program.

Special consideration must be given when a CALL statement with the BY CONTENT phrase is used to pass pointers and group items containing pointers. This is similar to the case of a MOVE statement. For a CALL BY CONTENT, an implicit MOVE of an item is done to create it in a temporary area. If the compiler can determine the offset of an item relative to a 16-byte boundary, that same offset is used when the implicit MOVE of the BY CONTENT item is done into the temporary area. When the compiler cannot determine the offset of an item relative to a 16-byte boundary, the implicit MOVE of the BY CONTENT item is done into a temporary area that is aligned on a 16-byte boundary.

The compiler is not able to determine the offset of an item relative to a 16-byte boundary when the BY CONTENT item is:

- Reference modified with an unknown starting position, or
- Defined in the Linkage Section.

When an operand is reference-modified, the offset is the reference modification starting position minus one, plus the operand's offset within the data structure. When an operand is in the Linkage Section, its offset can be determined from the calling program.

To avoid pointer alignment problems, pass items by reference.

The following is an example of passing items containing pointers, where pointer integrity is maintained in some cases, and not in others.

```cobol
WORKING-STORAGE SECTION.
01 A.
   05 B PIC X(3).
   05 C.
      10 FILLER PIC X(13).
      10 D POINTER.

PROCEDURE DIVISION.

CALL "B" USING A C.
```

*Figure 97. Program A -- Main Program*
WORKING-STORAGE SECTION.

01 E.
   05 F PIC X(16).
   05 G POINTER.
77 K PIC S9(3) VALUE 8.

LINKAGE SECTION.

01 A.
   05 B PIC X(3).
   05 C.
      10 FILLER PIC X(13).
      10 D POINTER.
01 C2.
   05 FILLER PIC X(13).
   05 D2 POINTER.

PROCEDURE DIVISION USING A C2.

CALL "C" USING BY CONTENT
   A, C2, E(5:), E(K:), F.

Figure 98. Program B -- Subprogram

In the previous example, Program A passes two group items to Program B. 1 is an 01-level group item, with an offset of zero. 2 is an 05-level group item, and has an offset of 3. Because the items are passed by reference, pointer integrity is maintained for both group items A and C.

Program B passes five items to another program, C. The items are passed by content to Program C. Because they are passed by content, Program C receives a copy of the items, and pointer integrity is not maintained in all cases.

- 3 Because this item is defined in the Linkage Section, it has an unknown offset. The compiler assumes it is 16-byte aligned, and in this case, when A is passed, pointer integrity of D is maintained, but a compiler warning message is issued on the CALL.
- 4 This item contains a pointer, and a pointer move is accomplished by 5. However, because the item is defined in the Linkage Section and the offset is unknown, pointer integrity is not maintained. The compiler attempts to move C2 to a 16-byte aligned area, and a compiler warning message is issued.
- 6 Because E contains a pointer, a pointer move is accomplished. The offset can be calculated because the reference modification start position is a numeric literal. In this case, pointer integrity is maintained, and the item is placed at an offset of 4 from the 16-byte boundary.
- 7 Because E contains a pointer, a pointer move is attempted. Because E is reference-modified with an unknown starting position (K), the compiler cannot calculate the offset, and assumes it is aligned on a 16-byte boundary. A compiler warning message is issued. If the value of K causes E to be aligned on a 16-byte boundary, pointer integrity is maintained. For this to occur, K must be 1 or 17.
- 8 F is an item defined in the Working-Storage Section, and contains no pointers, so no pointer moves are expected.
Using Pointers and APIs to Access User Spaces

The following example shows how you can use pointers to access user spaces and to chain records together.

POINTA is a program that reads customer names and addresses into a user space, and then displays the information in a list. The program assumes that the customer information exists in a file called POINTACU.

The customer address field is a variable-length field, to allow for lengthy addresses.

```
A+ THIS IS THE CUSTOMER INFORMATION FILE - POINTACUST
A
A
A  R FSCUST               TEXT('CUSTOMER MASTER RECORD')
A  FS_CUST_NO  8S00       TEXT('CUSTOMER NUMBER')
A                          ALIAS(FS_CUST_NUMBER)
A  FS_CUST_NM  20         TEXT('CUSTOMER NAME')
A                          ALIAS(FS_CUST_NAME)
A  FS_CUST_AD  100        TEXT('CUSTOMER ADDRESS')
A                          ALIAS(FS_CUST_ADDRESS)
A                          VARLEN
```

Figure 99. Example Using Pointers to Access User Spaces -- DDS
Figure 100 (Part 1 of 7). Example Using Pointers to Access User Spaces
Figure 100 (Part 2 of 7). Example Using Pointers to Access User Spaces
STMT SEQNBR -A 1 B..+....2....+....3....+....4....+....5....+....6....+....7.
IDENTFCN S COPYNAME CHG DATE

001040+ total length of this record including filler bytes
001050+ to make sure next record on 16 byte boundary

94 001060 10 ls-cust-address-length PIC S9(4) BINARY.
95 001070 05 ls-cust-address-data PIC X(116).

97 001200 MOVE "Error XX on file pointacu" TO ws-error-msg.
98 001210 MOVE ws-file-status TO ws-error-msg(7:2).

99 001260 SET ws-prog-loop to TRUE.
100 001270 PERFORM initial-display THRU read-initial-display
101 001280 UNTIL NOT ws-prog-loop.
102 001340 PERFORM main-loop UNTIL ws-prog-end
103 001350 MOVE 1 TO ws-current-rec
104 001360+ set ptr to header record
105 001390+ set to first customer record in space

110 001480 DISPLAY "Create Customer Information Area" AT 01B WITH
111 001490 BLANK SCREEN REVERSE-VIDEO
112 001550 MOVE spaces TO ws-error-msg
113 001560 END-IF.
114 001580 ACCEPT ws-accept-data AT 1056 WITH REVERSE-VIDEO
115 001590 ON EXCEPTION

116 001600 IF ws-status-1-func-key THEN
117 001620 SET ws-prog-end TO TRUE

118 001640 MOVE "Invalid Function Key" TO ws-error-msg
119 001670 MOVE "Unknown Error" TO ws-error-msg

Figure 100 (Part 3 of 7). Example Using Pointers to Access User Spaces
Customer Information Display


120 001700 IF ws-acc-create-space THEN
121 001710 PERFORM create-space THRU get-space
122 001720 SET ws-prog-continue TO TRUE
123 001730 ELSE
124 001740 IF NOT ws-acc-no-space THEN
125 001750 MOVE "Invalid Character Entered" TO ws-error-msg
126 001760 ELSE
127 001770 SET ws-prog-continue TO TRUE
128 001780 PERFORM get-space
129 001790 END-IF
130 001800 END-IF
131 001810 END-ACCEPT.
132 001820#create-space.
133 001830 CALL "QUSCRTUS" USING ws-space, ws-space-ptr.
134 001840 OPEN INPUT cust-file.
135 001850 IF ws-file-stat-good THEN
136 001860 READ cust-file AT END CONTINUE
137 001870 END-READ
138 001880 PERFORM VARYING ls-record-counter FROM 1 BY 1 UNTIL not ws-file-stat-good
139 001890 SET ls-cust-prev-ptr TO ws-cust-prev-ptr
140 001900 MOVE fs-cust-name TO ls-cust-name
141 001910 MOVE fs-cust-number TO ls-cust-number
142 001920 MOVE fs-cust-address-length TO ls-cust-address-length
143 001930 MOVE fs-cust-address-data(1:fs-cust-address-length) TO ls-cust-address-data(1:ls-cust-address-length)
144 001940 SET ADDRESS OF ls-user-space TO ADDRESS OF ls-user-space(ls-cust-rec-length + 1:1)
145 001950 ADD LENGTH OF ls-customer-rec BY 16 GIVING ws-temp
146 001960 DIVIDE ws-temp BY 16 GIVING ws-temp-z
147 001970 SUBTRACT ws-temp-2 FROM 16 GIVING ws-temp
148 001980 SAVE total record length in user space
149 001990 GET next record from file
150 002000 READ cust-file AT END CONTINUE
151 002010 END-READ
152 002020 END-PERFORM
153 002030 AT the end of the loop have one more record than really have
154 002040 SUBTRACT 1 FROM ls-record-counter
155 002050 CLOSE cust-file.
156 002060 END.
157 002070 main-loop.
158 002080 read-customer-file.
159 002090 read all records from customer file and move into space
160 002100 OPEN INPUT cust-file.
161 002110 IF ws-file-stat-good THEN
162 002120 READ cust-file AT END CONTINUE
163 002130 END-READ
164 002140 PERFORM VARYING ls-record-counter FROM 1 BY 1 UNTIL not ws-file-stat-good
165 002150 SET ls-cust-prev-ptr TO ws-cust-prev-ptr
166 002160 MOVE information from file into space
167 002170 Move info from file into space
168 002180 SET ws-cust-prev-ptr TO ADDRESS OF ls-user-space
169 002190 Make sure next record on 16 byte boundary
170 002200 ADD LENGTH OF ls-customer-rec BY 16 GIVING ws-temp
171 002210 DIVIDE ws-temp BY 16 GIVING ws-temp-z
172 002220 SUBTRACT ws-temp-2 FROM 16 GIVING ws-temp
173 002230 SAVE total record length in user space
174 002240 GET next record from file
175 002250 READ cust-file AT END CONTINUE
176 002260 END-READ
177 002270 END-PERFORM
178 002280 At the end of the loop have one more record than really have
179 002290 SUBTRACT 1 FROM ls-record-counter
180 002300 CLOSE cust-file.
181 002310 main-loop.
182 002320 write the records to the display until F3 entered

Figure 100 (Part 4 of 7). Example Using Pointers to Access User Spaces
Customer Information Display

5763C81 V3R9MS  001000  AS/400 COBOL Source  TESTER/POINTA  AS400SYS  05/01/94 18:01:14  Page 6

153 002410 DISPLAY "Customer Information" AT 0124 WITH 002420 BLANK SCREEN REVERSE-VIDEO 002430 "Cust Customer Name  Customer" 002440 "Address" 002450 "Number" AT 0405 002460 "F3=Exit" AT 2202.
002480 IF a pending error put on the display
154 002490 IF ws-error-msg NOT = SPACES THEN 002500 DISPLAY ws-error-msg at 2302 with beep highlight 002510 MOVE SPACES TO ws-error-msg 002520 END-IF.
002530 IF in the middle of the list put F7 on the display
155 002540 IF ws-current-rec > 1 THEN 23 002550 DISPLAY "F7=Back" AT 2240 002560 END-IF.
002570 save the current record
156 002580 MOVE ws-current-rec TO ws-old-rec.
160 002590 SET ws-old-space-ptr TO ADDRESS OF ls-user-space.
002600 move each record to the display
161 002610 PERFORM VARYING ws-line FROM ws-start-line BY 1 UNTIL ws-line > ws-displayed-lines or ws-current-rec > ls-record-counter
002620 IF ws-current-rec > 1 THEN 40 THEN
162 002650 MOVE "+" TO ws-plus
163 002660 MOVE "+" TO ws-plus
164 002670 MOVE 4 TO ws-temp-size
165 002680 ELSE
166 002690 MOVE ws-temp-size TO ws-temp-size
167 002700 MOVE SPACE TO ws-plus
168 002710 END-IF
169 002720 DISPLAY ls-cust-number at line ws-line column 5
170 002730 ls-cust-name ls-cust-address-data with size ws-temp-size ws-plus at line ws-line column 78
171 002740 get next record in the space
168 002770 ADD 1 TO ws-current-rec
169 002780 SET ADDRESS OF ls-user-space TO ADDRESS OF ls-cust-rec-length + 1:1
170 002790 END-IF
002810 PERFORM.
002820 IF can go forward put F8 on the display
171 002830 IF ws-current-rec < ls-record-counter THEN 23
172 002840 DISPLAY "F8=Forward" AT 2250 002850 END-IF.
002860 check to see if continue, exit, or get next records or previous records
172 002880 ACCEPT ws-accept-data WITH SECURE 25
002890 ON EXCEPTION
173 002900 IF ws-status-1-func-key THEN 174 002910 IF ws-func-03 THEN
175 002920 SET ws-prog-end TO TRUE 002930 ELSE
176 002940 IF ws-func-07 THEN
177 002950 PERFORM back-screen 002960 ELSE
178 002970 IF ws-func-08 THEN
179 002980 PERFORM forward-screen 002990 ELSE
180 003000 MOVE "Invalid Function Key" TO ws-error-msg
181 003010 MOVE ws-old-rec TO ws-current-rec
182 003020 SET ADDRESS OF ls-user-space TO ws-old-space-ptr 003030 END-IF
003040 END-IF
003050 ELSE
183 003060 MOVE "Unknown Error" TO ws-error-msg
184 003070 MOVE ws-old-rec TO ws-current-rec
185 003080 SET ADDRESS OF ls-user-space TO ws-old-space-ptr
003090 END-IF
003100 NOT ON EXCEPTION
186 003110 MOVE ws-old-rec TO ws-current-rec
187 003120 SET ADDRESS OF ls-user-space TO ws-old-space-ptr 003130 END-ACCEPT.
003140 do clean up for program

Figure 100 (Part 5 of 7). Example Using Pointers to Access User Spaces
Customer Information Display

003180+ keep reading end display until entered data correct
188 003170 SET ws-prog-loop to TRUE.
189 003180 PERFORM end-display THRU read-end-display 26
003190 UNTIL not ws-prog-loop.
003200 end-display.
190 003210 DISPLAY "Delete Customer Information Area" AT 0118 WITH 27
003220 BLANK SCREEN REVERSE-VIDEO
003230 "Delete customer information area (Y/N)=> <<
003240 AT 0115
003250 "F3=Exit" AT 2202.
003260 IF ws-error-msg NOT = SPACES THEN
003270 DISPLAY ws-error-msg at 2302 with beep highlight
003280 MOVE SPACES TO ws-error-msg
003290 END-IF.
003300 end-display.
194 003310 ACCEPT ws-accept-data AT 1056 WITH REVERSE-VIDEO
003320 ON EXCEPTION
195 003330 IF ws-status-1-func-key THEN
196 003340 IF ws-func-03 THEN
197 003350 SET ws-prog-end TO TRUE
003360 ELSE
003370 MOVE "Invalid Function Key" TO ws-error-msg
003380 END-IF
003390 ELSE
003400 MOVE "Unknown Error" TO ws-error-msg
003410 END-IF
003420 NOT ON EXCEPTION
200 003430 IF ws-acc-delete-space THEN
201 003440 PERFORM delete-space
202 003450 SET ws-prog-continue TO TRUE
003460 ELSE
003470 IF NOT ws-acc-no-space THEN
003480 MOVE "Invalid Character Entered" TO ws-error-msg
003490 ELSE
003500 SET ws-prog-continue TO TRUE
003510 END-IF
003520 END-IF
003530 END-ACCEPT.
003540 back-screen. 28
206 003550 IF ws-old-rec <= 1 THEN
207 003560 MOVE "Top of customer records" TO ws-error-msg
208 003570 MOVE ws-old-rec TO ws-current-rec 29
209 003580 SET ADDRESS OF ls-user-space TO ws-old-space-ptr
003590 ELSE
003600 MOVE ws-old-rec TO ws-current-rec 29
210 003610 SET ADDRESS OF ls-user-space TO ws-old-space-ptr
211 003620 PERFORM VARYING ws-line FROM ws-start-line BY 1
003630 UNTIL ws-line > ws-displayed-lines or
212 003640 ws-current-rec <= 1
003650 BACK UP ONE RECORD AT A TIME
213 003660 SET ws-cust-prev-ptr TO ls-cust-prev-ptr
214 003670 SET ADDRESS OF ls-user-space TO ws-cust-prev-ptr 30
215 003680 SUBTRACT FROM ws-current-rec
003690 END-PERFORM
003700 END-IF.
003710 forward-screen. 28
003720 IF current record greater or equal to the max records
003730 print error, have reached max records
216 003740 IF ws-current-rec > ls-record-counter
217 003750 MOVE "No more customer records" TO ws-error-msg
218 003760 MOVE ws-old-rec TO ws-current-rec
219 003770 SET ADDRESS OF ls-user-space TO ws-old-space-ptr
003780 ELSE
220 003790 MOVE ws-current-rec TO ws-old-rec
221 003800 SET ws-old-space-ptr TO ADDRESS OF ls-user-space
003810 END-IF.

Chapter 12. Communicating Between Programs 297
### Message Summary

<table>
<thead>
<tr>
<th>Total</th>
<th>Info(0-4)</th>
<th>Warning(5-19)</th>
<th>Error(20-29)</th>
<th>Severe(30-39)</th>
<th>Terminal(40-99)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Source records read . . . . . . . . : 381  
Copy records read . . . . . . . . . : 15  
Copy members processed . . . . . . : 1  
Sequence errors . . . . . . . . . . : 0  
Highest severity message issued . . : 0

LBL901 DD Program POINTA created in library TESTER.

**** END OF COMPILATION ****

---

**Figure 100 (Part 7 of 7). Example Using Pointers to Access User Spaces**

1. The compiler directive TITLE is used to create this title that appears at the beginning of each page.

2. CRT STATUS IS specifies a data name into which a status value is placed after the termination of an extended ACCEPT statement. In this example, the STATUS key value is used to determine which function key was pressed.

3. The CRT STATUS as mentioned in 2 is defined here.

4. `fs-cust-address` is a variable-length field. To see meaningful names here rather than FILLER, specify `*VARCHAR` for the CVTOPT parameter of the CRTCBBLPGM command, or VARCHAR in the PROCESS statement, as shown in 2. For more information about variable-length fields, refer to “Declaring Data Items Using CVTOPT Data Types” on page 130.

5. The `ws-params` structure contains the parameters used when calling the APIs to access user spaces.

6. `ws-err-data` is the structure for the error parameter for the user space APIs. Note that the `ws-input-l` is zero, meaning that any exceptions are signalled to the program, and not passed in the error code parameter. For more information on error code parameters, refer to the System Programmer’s Interface Reference.

7. `ws-space-ptr` defines a pointer data item set by the API QUSPTRUS. This points to the beginning of the user space, and is used to set the addresses of items in the Linkage Section.

8. The first data structure (`ls-header-record`) to be defined in the user space.

9. FILLER is used to maintain pointer alignment, because it makes `ls-header-record` a multiple of 16 bytes long.

10. The second data structure (`ls-user-space`) to be defined in the user space.

11. `initial-display` shows the Create Customer Information Area display. This display is shown in Figure 101 on page 300.

12. `read-initial-display` reads the first display, and determines if the user chooses to continue or end the program. If the user continues the program by pressing Enter, then the program checks `ws-accept-data` to see if the customer information area is to be created.

13. `QUSCRTUS` is an API used to create user spaces.
QUSPTRUS is an API used to return a pointer to the beginning of a user space.

Maps the first data structure (*ls-header-record*) over the beginning of the user space.

Maps the second data structure (*ls-user-space*) over the beginning of the user space.

Uses ADDRESS OF special register

Uses ADDRESS OF, not the ADDRESS OF special register, because it is reference modified.

QUSDLTUS is an API used to delete a user space.

The following four arithmetic statements calculate the total length of each record, and ensure that each record is a multiple of 16 bytes in length.

*main-loop* puts up the Customer Information display. Refer to Figure 102 on page 300.

These statements determine if the program should display function keys F7 and F8.

Saves a pointer to the first customer record on the display.

This ACCEPT statement waits for input from the Customer Information display. Based on the function key pressed, it calls the appropriate paragraph to display the next set of records (*forward-screen*), or the previous set of records (*back-screen*), or sets an indicator to end the routine if F3 is pressed.

The clean up routine displays the Delete Customer Information Area display until an appropriate key is pressed.

This statement puts up the Delete Customer Information Area display.

Each record contains a pointer to the previous customer record. The ADDRESS OF special register points to the current customer record. By changing the ADDRESS OF special register, the current customer record is changed.

*back-screen* moves the current record pointer backward one record at a time, by moving the pointer to the previous customer record into the pointer to the current customer record (ADDRESS OF). Before moving backward one record at a time, the program sets the current customer record to the first record currently displayed.

*forward-screen* sets *ws-old-space-ptr* (which points to the first record in the display) to point to the current record (which is after the last record displayed.)

A user space always begins on a 16-byte boundary, so the method illustrated here ensures that all records are aligned. *ls-cust-rec-length* is also used to chain the records together.
When you run POINTA, you see the following displays:

![Create Customer Information Area Display](image)

Create customer information area (Y/N) => y <=

F3=Exit

Figure 101. Create Customer Information Area Display

If you specify Y to create the user space, the program reads the customer records from the file and puts the information in the user space. The records are chained together.

When you press enter from the previous display, the Customer Information display appears:

![Customer Information Display](image)

Customer Information

<table>
<thead>
<tr>
<th>Cust Number</th>
<th>Customer Name</th>
<th>Customer Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>Bakery Unlimited</td>
<td>30 Bake Way, North York</td>
</tr>
<tr>
<td>00000002</td>
<td>Window World</td>
<td>150 Eglinton Ave E., North York, Ontario</td>
</tr>
<tr>
<td>00000003</td>
<td>Jons Clothes</td>
<td>101 Park St, North Bay, Ontario, Canada</td>
</tr>
<tr>
<td>00000004</td>
<td>Pizza World</td>
<td>254 Main Street, Toronto, Ontario</td>
</tr>
<tr>
<td>00000005</td>
<td>Marvin's Auto Body</td>
<td>9 George St, Peterborough, Ontario, Canada</td>
</tr>
<tr>
<td>00000006</td>
<td>Jack's Snacks</td>
<td>23 North St, Timmins, Ontario, Canada</td>
</tr>
<tr>
<td>00000007</td>
<td>Video World</td>
<td>14 Robson St, Vancouver, B.C, Canada</td>
</tr>
<tr>
<td>00000008</td>
<td>Pat's Daycare</td>
<td>8 Kingston Rd, Pickering, Ontario, Canada</td>
</tr>
<tr>
<td>00000009</td>
<td>Mary's Pies</td>
<td>3 Front St, Toronto, Ontario, Canada</td>
</tr>
<tr>
<td>00000010</td>
<td>Carol's Fashions</td>
<td>19 Spark St, Ottawa, Ontario, Canada</td>
</tr>
<tr>
<td>00000011</td>
<td>Grey Optical</td>
<td>5 Lundy's Lane, Niagara Falls, Ont. Cana</td>
</tr>
<tr>
<td>00000012</td>
<td>Fred's Forage</td>
<td>33 Dufferin St, Toronto, Ontario, Canada</td>
</tr>
<tr>
<td>00000013</td>
<td>Dave's Trucking</td>
<td>15 Water St, Guelph, Ontario, Canada</td>
</tr>
<tr>
<td>00000014</td>
<td>Doug's Music</td>
<td>101 Queen St, Toronto, Ontario, Canada</td>
</tr>
<tr>
<td>00000015</td>
<td>Anytime Copiers</td>
<td>300 Warden Ave, Scarborough, Ontario, Ca</td>
</tr>
<tr>
<td>00000016</td>
<td>Rosa's Ribs</td>
<td>440 Avenue Rd, Toronto, Ontario, Canada</td>
</tr>
</tbody>
</table>

F3=Exit F8=Forward

Figure 102. Customer Information Area Display

If there are more than 16 records in the user space (based on the starting line in `ws-start-line`), the program enables the F8=Forward key, to allow the user to page
forward in the list. Once the user has rolled forward, the F7=Backward key is enabled to allow the user to page backward in the list, as shown in the following display:

<table>
<thead>
<tr>
<th>Cust Number</th>
<th>Customer Name</th>
<th>Customer Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000017</td>
<td>Picture It</td>
<td>33 Kingston Rd, Ajax, Ontario, Canada</td>
</tr>
<tr>
<td>00000018</td>
<td>Paula's Flowers</td>
<td>144 Pape Ave, Toronto, Ontario, Canada</td>
</tr>
<tr>
<td>00000019</td>
<td>Mom's Diapers</td>
<td>101 Ford St, Toronto, Ontario, Canada</td>
</tr>
<tr>
<td>00000020</td>
<td>Chez Francois</td>
<td>1202 Rue Ste Anne, Montreal, PQ, Canada</td>
</tr>
<tr>
<td>00000021</td>
<td>Vetements de Louise</td>
<td>892 Rue Sherbrooke, Montreal E, PQ, Canada</td>
</tr>
<tr>
<td>00000022</td>
<td>Good Eats</td>
<td>355 Lake St, Port Hope, Ontario, Canada</td>
</tr>
</tbody>
</table>

**Figure 103. Customer Information Display (Second Display)**

When the user exits from the above display, the option to delete the user space is presented, as shown in the following display:

**Figure 104. Delete Customer Information Display**
Processing a Chained List

A typical application for using pointer data items is in processing a chained list (a series of records where each one points to the next).

For this example, picture a chained list of data that is composed of individual salary records. Figure 105 shows one way to visualize how these records are linked in storage:

![Diagram of a Chained List]

Figure 105. Representation of a Chained List Ending with NULL

The first item in each record (except for the last record) points to the next record. The first item in the last record, in order to indicate that it is the last record, contains a null value instead of an address.

The high-level logic of an application that processes these records might look something like this:

```
OBTAIN ADDRESS OF FIRST RECORD IN CHAINED LIST FROM ROUTINE
CHECK FOR END OF THE CHAINED LIST
DO UNTIL END OF THE CHAINED LIST
  PROCESS RECORD
  GO ON TO THE NEXT RECORD
END
```

Figure 106 on page 303 contains an outline of the processing program, LISTS, used in this example of processing a chained list.
IDENTIFICATION DIVISION.
PROGRAM-ID. LISTS.
ENVIRONMENT DIVISION.
DATA DIVISION.
*****
WORKING-STORAGE SECTION.
77 PTR-FIRST POINTER VALUE IS NULL.
77 DEPT-TOTAL PIC 9(4) VALUE IS 0.
*****
LINKAGE SECTION.
01 SALARY-REC.
  02 PTR-NEXT-REC POINTER.
  02 NAME PIC X(26).
  02 DEPT PIC 9(4).
  02 SALARY PIC 9(6).
  01 DEPT-X PIC 9(4).
*****
PROCEDURE DIVISION USING DEPT-X.
*****
* FOR EVERYONE IN THE DEPARTMENT RECEIVED AS DEPT-X,
* GO THROUGH ALL OF THE RECORDS IN THE CHAINED LIST BASED ON THE
* ADDRESS OBTAINED FROM THE PROGRAM CHAIN-ANCH
* AND ACCUMULATE THE SALARIES.
* IN EACH RECORD, PTR-NEXT-REC IS A POINTER TO THE NEXT RECORD
* IN THE LIST; IN THE LAST RECORD, PTR-NEXT-REC IS NULL.
* DISPLAY THE TOTAL.
*****
CALL "CHAIN-ANCH" USING PTR-FIRST
SET ADDRESS OF SALARY-REC TO PTR-FIRST
*****
PERFORM WITH TEST BEFORE UNTIL ADDRESS OF SALARY-REC = NULL
  IF DEPT = DEPT-X
    THEN ADD SALARY TO DEPT-TOTAL
  ELSE CONTINUE
END-IF
SET ADDRESS OF SALARY-REC TO PTR-NEXT-REC
END-PERFORM
*****
DISPLAY DEPT-TOTAL
GOBACK.

Figure 106. Program for Processing a Chained List

Passing Addresses between Programs
To obtain the address of the first SALARY-REC record area, the LISTS program
calls the program CHAIN-ANCH:

CALL "CHAIN-ANCH" USING PTR-FIRST

PTR-FIRST is defined in WORKING-STORAGE in the calling program (LISTS) as a
pointer data item:

WORKING-STORAGE SECTION.
77 PTR-FIRST POINTER VALUE IS NULL.

Upon return from the call to CHAIN-ANCH, PTR-FIRST contains the address of the
first record in the chained list.

PTR-FIRST is initially defined as having a null value as a logic check. If an error
occurs with the call, and PTR-FIRST never receives the value of the address of the
first record in the chain, a null value remains in PTR-FIRST and, according to the
logic of the program, the records will not be processed.
NULL is a figurative constant used to assign the value of a non-valid address to pointer items. It can be used in the VALUE IS NULL clause, in the SET statement, and as an operand in a relation condition with a pointer data item.

The Linkage Section of the calling program contains the description of the records in the chained list. It also contains the description of the department code that is passed through the USING phrase of the CALL statement.

```
LINKAGE SECTION.
  01 SALARY-REC.
    02 PTR-NEXT-REC   POINTER.
    02 NAME            PIC X(20).
    02 DEPT            PIC 9(4).
    02 SALARY          PIC 9(6).
  01 DEPT-X           PIC 9(4).
```

To “base” the record description SALARY-REC on the address contained in PTR-FIRST, use a SET statement:
```
CALL "CHAIN-ANCH" USING PTR-FIRST
SET ADDRESS OF SALARY-REC TO PTR-FIRST
```

**Check for the End of the Chained List**

The chained list in this example is set up so that the last record contains a non-valid address. To do this, the pointer data item in the last record would be assigned the value NULL.

A pointer data item can be assigned the value NULL in two ways:

- A pointer data item can be defined with a VALUE IS NULL clause in its data definition.
- NULL can be the sending field in a SET statement.
- The initial value of a pointer data item with or without a VALUE clause of NULL equals NULL.

In the case of a chained list in which the pointer in the last record contains a null value, the code to check for the end of the list would be:
```
IF PTR-NEXT-REC = NULL
:

(logic for end of chain)
```

If you have not reached the end of the list, process the record and move on to the next record.

In the program LISTS, this test for the end of the chained list is accomplished with a “do while” structure:
```
PERFORM WITH TEST BEFORE UNTIL ADDRESS OF SALARY-REC = NULL
  IF DEPT = DEPT-X
    THEN ADD SALARY TO DEPT-TOTAL
  ELSE CONTINUE
END-IF
SET ADDRESS OF SALARY-REC TO PTR-NEXT-REC
END-PERFORM
```
Continuing Processing the Next Record

To move on to the next record, set the address of the record in the Linkage Section to be equal to the address of the next record. This is accomplished through the pointer data item sent as the first field in SALARY-REC:

```
SET ADDRESS OF SALARY-REC TO PTR-NEXT-REC
```

Then repeat the record-processing routine, which will process the next record in the chained list.

Incrementing Addresses Received from Another Program

The data passed from a calling program might contain header information that you want to ignore (for example, in data received from a CICS application that is not migrated to the command level).

Because pointer data items are not numeric, you cannot directly perform arithmetic on them. You can, however, use the SET verb to increment the passed address in order to bypass header information.

You could set up the Linkage Section as follows:

```
LINKAGE SECTION.
  01 RECORD-A.
    02 HEADER PIC X(16).
    02 REAL-SALARY-REC PIC X(30).
  01 SALARY-REC.
    02 PTR-NEXT-REC POINTER.
    02 NAME PIC X(20).
    02 DEPT PIC 9(4).
    02 SALARY PIC 9(6).
```

Within the Procedure Division, base the address of SALARY-REC on the address of REAL-SALARY-REC:

```
SET ADDRESS OF SALARY-REC TO ADDRESS OF REAL-SALARY-REC
```

SALARY-REC is now based on the address of RECORD-A + 16.

Data Areas

A data area is an object used to communicate data such as variable values between programs within a job and between jobs. A data area can be created and declared to a program before it is used in that program or job. For information on how to create and declare a data area, see the CL Programmer's Guide.

Local Data Area

The local data area can be used to pass any desired information between programs in a job. This information may be free-form data, such as informal messages, or may consist of a fully structured or formatted set of fields.

The system automatically creates a local data area for each job. The local data area is defined outside the COBOL program as an area of 1024 bytes.
When a job is submitted, the submitting job's local data area is copied into the submitted job's local data area. If there is no submitting job, the local data area is initialized to blanks.

A COBOL program can access the local data area for its job with the ACCEPT and DISPLAY statements, using a mnemonic name associated with the function-name LOCAL-DATA.

There is only one local data area associated with each job. Even if several work stations are acquired by a single job, only one local data area exists for that job. There is not a local data area for each work station.

**Program Initialization Parameters (PIP) Data Area**

The PIP data area is used by a prestart job. Generally, a prestart job is a job from a remote system under ICF that you start and keep ready to run until you call it.

If you use a prestart job, you do not have to wait for a program that you call to go through job initiation processing. Job initiation is performed before a program can actually start. Because job initiation has already taken place, a prestart job allows your program to start more quickly after the program start request is received.

A COBOL program can access the PIP data area for its job with the ACCEPT statement, using a mnemonic name associated with the function-name PIP-DATA.

The PIP data area is a 2000-byte alphanumeric item and contains parameters received from a calling program. It provides the program initialization parameters that, in non-prestart jobs, is provided through standard COBOL parameters.

You use a Format 5 ACCEPT statement to access the PIP data area, similar to the way in which you use a Format 4 ACCEPT statement to read from the local data area. Note that you cannot update the PIP data area using COBOL. See the COBOL/400 Reference for detailed syntax information.

For more information regarding prestart jobs and the PIP data area, refer to the Work Management Guide and the CL Programmer’s Guide.

**File Considerations**

You can pass a file name as a parameter in a COBOL program, but you cannot use that file in the called program. If a file is defined in both a calling program and a called program, it is treated as two separate files. The contents of the record area and the current record pointer in each program are independent, unless shared files are specified in CL commands. See the Data Management Guide for further information on shared files.

The following statements affect file status differently:

- An EXIT PROGRAM statement does not change the status of any of the files in a run unit.
- A STOP RUN statement closes all of the files in a run unit.
A GOBACK statement issued from a main program closes all of the files in a run unit. A GOBACK statement issued from a subprogram does not change the status of any of the files in a run unit.

A CANCEL statement does not change the status of any of the files in the program that is canceled. It does free the storage that contains information about the file. If the program has files that are open when the CANCEL statement is processed, those files are closed when that program is cancelled. The program can no longer use the file. If the canceled program is called again, the program considers the file closed. If the program opens the file, a new linkage to the file is established. This can cause additional system storage to be used.
Appendix A. Segmentation Feature

You do not have to be concerned with storage management when writing COBOL/400 programs. Storage segmentation is, however, available for compatibility with other systems.

The segmentation feature provides programmer-controlled storage optimization of the Procedure Division by allowing that division to be subdivided both physically and logically.

Segmentation Concepts

Although it is not required, the Procedure Division of a source program is often written as a consecutive group of sections, each of which is made up of a series of related operations that perform a particular function. Thus, the entire Procedure Division is made up of a number of logical subdivisions. Segmentation allows the programmer to physically divide the Procedure Division into segments, each of which has specific physical and logical attributes.

When Segmentation is used, the entire Procedure Division must be divided into sections. Each section must then be classified as to its physical and logical attributes. Classification is specified by means of segment numbers. All sections given the same segment number make up one program segment.

Segment numbers must be integers from 0 through 99.

Program Segments

There are three types of program segments; fixed permanent, fixed overlayable, and independent.

Fixed Segments

Fixed-permanent segments and fixed-overlayable segments make up the fixed portion, the part of the Procedure Division that is logically treated as if it were always physically present in main storage. Fixed-portion segment numbers must be integers from 0 through 49.

A fixed-permanent segment is always made available in its last-used state.

A fixed-overlayable segment is logically always in main storage during program processing; therefore, it is always available in its last-used state. Any overlay of such a segment is transparent to the user. Thus, a fixed-overlayable segment is logically identical with a fixed-permanent segment.

Independent Segments

Logically, an independent segment can overlay and be overlaid by other segments during a program’s run.

An independent segment is made available in its initial state the first time control is passed to it (explicitly or implicitly) during a program’s run.
An independent segment is made available in its initial state during subsequent transfers of control when:

- The transfer is the result of an implicit transfer of control between consecutive statements that are in different segments (that is, when control drops through into the independent segment from the physically preceding segment).
- The transfer is the result of an implicit transfer from a SORT or MERGE statement in one segment to a SORT input procedure or SORT/MERGE output procedure in an independent segment.
- An explicit transfer of control from a section with a different segment number takes place (as, for example, during the transfer of control in a PERFORM n TIMES statement).

An independent segment is made available in its last-used state during subsequent transfers of control when:

- With the exception of the two preceding kinds of implied transfers, an implicit transfer from a section with a different priority takes place (as, for example, when control is returned to the independent segment from a Declarative procedure).
- An explicit transfer results from an EXIT PROGRAM or GOBACK statement.

Independent segments must be assigned segment numbers 50 through 99.

**Segmentation Logic**

In a segmented program, the sections are classified by a system of segment numbers according to the following criteria:

- **Frequency of Reference**—Much-referenced sections, or those that must be available for reference at all times, should be placed within fixed permanent segments. Less frequently used sections can be within either fixed overlayable or independent segments, depending on the program logic.
- **Frequency of Use**—The more frequently a section is used, the lower its segment number; the less frequently it is referred to, the higher its segment number.
- **Logical Relationships**—Sections that frequently communicate with each other should be given identical segment numbers.

**Segmentation Control**

Except for specific transfers of control, the logical sequence and the physical sequence of program instructions are the same. The compiler inserts any instructions necessary to initialize a segment. It is not necessary to transfer control to the beginning of a segment, or to the beginning of a section within a segment. Instead, control can be transferred to any paragraph in the Procedure Division.

**COBOL Source Program Considerations**

The following elements of a COBOL source program implement the Segmentation feature:

- The SEGMENT-LIMIT clause in the OBJECT-COMPUTER paragraph of the Environment Division. This clause allows you to control the specification of fixed-permanent and fixed-overlayable segments.
• Procedure Division segment numbers, which group sections into segments. The segment numbering scheme also allows specifications of independent segments, fixed-permanent segments, and (in conjunction with the SEGMENT-LIMIT clause) of fixed-overlayable segments.

**Segmentation–Environment Division**
In the OBJECT-COMPUTER paragraph, the SEGMENT-LIMIT clause allows the user to reclassify fixed permanent segments while retaining the properties of fixed portion segments for the reclassified segments.

<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEGMENT-LIMIT IS segment-number .</td>
</tr>
</tbody>
</table>

The SEGMENT-LIMIT clause allows the programmer to specify certain permanent segments as capable of being overlaid by independent segments without losing the logical properties of fixed portion segments.

segment-number must be an integer ranging in value from 1 through 49.

When the SEGMENT-LIMIT clause is specified:
• Fixed-permanent segments are those with segment numbers from 0 up to, but not including, the segment number specified.
• Fixed-overlayable segments are those with segment numbers from the segment number specified through 49.

For example, if SEGMENT-LIMIT IS 25 is specified, sections with segment numbers 0 through 24 are fixed-permanent segments, and sections with segment numbers 25 through 49 are fixed-overlayable segments.

When the SEGMENT-LIMIT clause is omitted, all sections with segment numbers 0 through 49 are fixed-permanent segments.

**Segmentation–Procedure Division**
In the Procedure Division of a segmented program, section classification is specified through segment numbers in the section headers. The segment number must be an integer from 0 through 99.

<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>section-name SECTION segment-number .</td>
</tr>
</tbody>
</table>

All sections with the same segment number make up one program segment. Such sections need not be contiguous in the source program.
Segments with segment numbers 0 through 49 are in the fixed portion of the program. Declarative sections can be assigned only these segment numbers. Segments with segment numbers from 50 through 99 are independent segments. If the segment number is omitted from the section header, the segment number is assumed to be 0.

**Segmentation—Special Considerations**

When segmentation is used, there are restrictions on the ALTER, PERFORM, SORT, and MERGE statements. There are also special considerations for calling and called programs.

**ALTER Statement**

A GO TO statement in an independent segment must not be referred to by an ALTER statement in a different segment. All other uses of the ALTER statement are valid and are performed, even if the GO TO statement referred to is in a fixed-overlayable segment.

**PERFORM Statement**

A PERFORM statement in the fixed portion can have in its range, in addition to any Declarative procedures, the processing of which is caused within that range, only one of the following:

- Sections and/or paragraphs in the fixed portion
- Sections and/or paragraphs contained within a single independent segment.

A PERFORM statement in an independent segment can have within its range, in addition to any Declarative procedures, the processing of which is caused within that range, only one of the following:

- Sections and/or paragraphs in the fixed portion
- Sections and/or paragraphs wholly contained in the same independent segment as the PERFORM statement.

**SORT and MERGE Statements**

If a SORT or MERGE statement appears in the fixed portion, any SORT input procedures or SORT/MERGE output procedures must appear completely in one of the following:

- The fixed portion
- A single independent segment.

If a SORT or MERGE statement appears in an independent segment, any SORT input procedures or SORT/MERGE output procedures must appear completely in one of the following:

- The fixed portion
- The same independent segment as the SORT or MERGE statement.

**Calling and Called Programs**

The CALL statement can appear anywhere within a segmented program. When a CALL statement appears in an independent segment, that segment is in its last-used state when control is returned to the calling program.
Appendix B. Debugging Features

The debugging features specify the conditions under which procedures are to be monitored during program run time.

COBOL source language debugging statements are provided. You must decide what to monitor, and what information you need to retrieve for debugging purposes. The COBOL debugging features simply provide access to pertinent information.

COBOL Source Language Debugging

COBOL language elements that implement the Debugging Feature are a compile-time switch (WITH DEBUGGING MODE), a run-time switch, a USE FOR DEBUGGING Declarative, the special register DEBUG-ITEM, and debugging lines that can be written in the Environment, Data, and Procedure Divisions.

Compile-Time Switch

In the SOURCE-COMPUTER paragraph of the Configuration Section, the WITH DEBUGGING MODE clause acts as a compile-time switch.

```
SOURCE-COMPUTER. ─────────────────────────────────────────────────────────────────
                       └─computer name ─────────────────────────┐
                       └─┬──────┬──DEBUGGING MODE─┘
                       └─WITH─┘
```

The WITH DEBUGGING MODE clause serves as a compile-time switch for the debugging statements written in the source program.

When WITH DEBUGGING MODE is specified, all debugging sections and debugging lines are compiled as specified in this appendix. When WITH DEBUGGING MODE is omitted, all debugging sections and debugging lines are treated as documentation.
Run-Time Switch

The run-time switch dynamically activates the debugging code that is generated when WITH DEBUGGING MODE is specified.

Two commands are provided to control the run-time switch. To set the run-time switch on, enter the command:

```
STRCBLDBG
```

and press F4.

You see the following display:

```
Start COBOL Debug (STRCBLDBG)
Type choices, press Enter.
Program . . . . . . . . . . . . Name
Library . . . . . . . . . . . *LIBL Name, *LIBL, *CURLIB
```

The following diagram shows the syntax of the STRCBLDBG command:

```
/SM590000/SM590000──STRCBLDBG──┬────────────────────────────────────────────┬─────────────────/SM590000/SM630000
└─PGM──(─┬─────────────────┬──
program-name
                         └─
```

```
├──/c5197LIBL/─────────┤
├──/c5197CURLIB/───────┤
└──library-name/──┘
```

```
```

Figure 107. Syntax of the STRCBLDBG Command

This command is allowed in interactive and batch processing, and in CL programs.

General-Use Programming Interface

You can use this command in QCMDEXC.

End of General-Use Programming Interface
To set the run-time switch off, enter the command:

```
ENDCBLDBG
```

and press F4.

You see the following display:

```
End COBOL Debug (ENDCBLDBG)
Type choices, press Enter.
Program . . . . . . . . . . . . Name
Library . . . . . . . . . . . +LIBL Name, +LIBL, +CURLIB
```

The following diagram shows the syntax of the ENDCBLDBG command:

```
/SM590000/SM590000 ──ENDCBLDBG──┬────────────────────────────────────────────┬─────────────────/SM590000/SM630000
└─PGM──(─┬─────────────────┬──program-name─)─┘
     ├──/c5197LIBL
     └──/c5197CURLIB/
           └──library-name/
```

Figure 108. Syntax of the ENDCBLDBG Command

This command is allowed in interactive and batch processing, and in CL programs.

General-Use Programming Interface

You can use this command in QCMDEXC.

End of General-Use Programming Interface

The default for the run-time switch is off.

When debugging mode is specified through the run-time switch, all the debugging sections and debugging lines (D in column 7) compiled into the program are activated.
You must enter the STRCBLDBG command for each COBOL program (main program or called program) to be debugged in the next COBOL run unit. At the end of the run unit, all run-time switches that are on are set off. If a switch must be set off before starting a COBOL run unit, use the ENDCBLDBG command. Run-time switches for up to 15 programs can be on at once.

When the STRCBLDBG or ENDCBLDBG command is issued in a CL program, concatenation expressions can be used for all parameter values. See the CL Programmer's Guide for more information about concatenation expressions.

When debugging mode is suppressed, through the run-time switch, any USE FOR DEBUGGING Declarative procedures are inhibited. All debugging lines (D in column 7) remain in effect.

Recompilation of the source program is not required to activate or deactivate the run-time switch.

When WITH DEBUGGING MODE is not specified in the SOURCE-COMPUTER paragraph, the run-time switch has no effect on the running of the program.

**USE FOR DEBUGGING Declarative**

The USE FOR DEBUGGING sentence in the Procedure Division identifies the items in the source program that are to be monitored by the associated debugging declarative procedure.

![Format Diagram]

Identifier-1 cannot be reference modified.

When specified, all debugging sections must be written immediately after the DECLARATIVES header. Except for the USE FOR DEBUGGING sentence there must be no reference to any non-declarative procedure within the debugging procedure.

Note that the USE FOR DEBUGGING declarative causes all subsequent statements to be ignored up to a valid USE AFTER EXCEPTION/ERROR statement, or END DECLARATIVES delimiter. Entire programs can be ignored because of this.

Automatic running of a debugging section is not caused by a statement appearing in a debugging section.
A debugging section for a specific operand is processed only once as the result of the running of a single statement, no matter how many times the operand is specified in the statement. An exception to this rule is that each specification of a subscripted or indexed identifier where the subscripts or indexes are different causes the calling of the debugging Declarative. For a PERFORM statement that causes repeated running of a procedure, any associated procedure name debugging Declarative section is run only once for each processing of the procedure.

For debugging purposes, each separate occurrence of an imperative verb within an imperative statement begins a separate statement.

Statements appearing outside the debugging sections must not refer to procedure names defined within the debugging sections.

Except for the USE FOR DEBUGGING sentence itself, statements within a debugging Declarative section can only refer to procedure names defined in a different USE procedure through the PERFORM statement. Procedure names within debugging Declarative sections must not appear in USE FOR DEBUGGING sentences.

Table 7 defines the points during program run time when the USE FOR DEBUGGING procedures are processed. Identifier-n, file-name-n, and procedure-name-n refer to the first and all subsequent specifications of that type of operand in one USE FOR DEBUGGING sentence. Any particular identifier, file name, or procedure name can appear in only one USE FOR DEBUGGING sentence, and only once in that sentence.

An identifier in a USE FOR DEBUGGING sentence:

- Must be specified without the subscripting or indexing normally required if it contains an OCCURS clause or is subordinate to an entry containing an OCCURS clause. (A SEARCH or SEARCH ALL statement that refers to such an identifier does not call the USE FOR DEBUGGING procedures.)
- Must not be a special register.

When ALL PROCEDURES is specified in a USE FOR DEBUGGING sentence, procedure-name-1, procedure-name-2, procedure-name-3, and so on, must not be specified in any USE FOR DEBUGGING sentence. The ALL PROCEDURES phrase can be specified only once in a program.

When a USE FOR DEBUGGING operand is used as a qualifier, such a reference in the program does not activate the debugging procedures.

References to the DEBUG-ITEM special register can be made only from within a debugging Declarative procedure.
<table>
<thead>
<tr>
<th>USE FOR DEBUGGING Operand</th>
<th>The USE FOR DEBUGGING procedures run immediately after the following:</th>
</tr>
</thead>
</table>
| identifier-n              | Before REWRITE/WRITE identifier-n and after FROM phrase move, if applicable.  
|                           | After each initialization, modification, or evaluation of identifier-n in PERFORM/VARYING/AFTER/UNTIL identifier-n.  
|                           | After any other COBOL statement that explicitly refers to identifier-n and could change its contents. (See note.) |
| ALL REFERENCES OF identifier-n | Before GO TO DEPENDING ON identifier-n, control is transferred, and before any associated debugging section for the procedure name runs.  
|                           | Before REWRITE/WRITE identifier-n and FROM phrase move, if applicable.  
|                           | After each initialization, modification or evaluation of identifier-n in PERFORM/VARYING/AFTER/UNTIL identifier-n.  
|                           | After any other COBOL statement explicitly referring to identifier-n. (See note.) |
| file-name-n               | After CLOSE/DELETE/OPEN/START file-name-n.  
|                           | After READ file-name-n where AT END/INVALID KEY was not run. |
| procedure-name-n          | Before each running of the named procedure.  
|                           | After running an ALTER statement referring to the named procedure. |
| ALL PROCEDURES            | Before each running of every non-debugging procedure.  
|                           | After running every ALTER statement (except ALTER statements in Declarative procedures). |

**Note:** Operands acted upon but not explicitly named in such statements as ADD, MOVE, or SUBTRACT CORRESPONDING never cause activation of a USE FOR DEBUGGING procedure when such statements are run. If identifier-n is specified in a phrase that is not processed, the associated debugging section is not run.
DEBUG-ITEM Special Register

The DEBUG-ITEM special register provides information for a debugging Declarative procedure. DEBUG-ITEM has the following implicit description:

```
01 DEBUG-ITEM.
  02 DEBUG-LINE       PICTURE IS X(6).
  02 FILLER          PICTURE IS X VALUE SPACE.
  02 DEBUG-NAME      PICTURE IS X(30).
  02 FILLER          PICTURE IS X VALUE SPACE.
  02 DEBUG-SUB-1     PICTURE IS S9999 SIGN IS LEADING SEPARATE CHARACTER.
  02 FILLER          PICTURE IS X VALUE SPACE.
  02 DEBUG-SUB-2     PICTURE IS S9999 SIGN IS LEADING SEPARATE CHARACTER.
  02 FILLER          PICTURE IS X VALUE SPACE.
  02 DEBUG-SUB-3     PICTURE IS S9999 SIGN IS LEADING SEPARATE CHARACTER.
  02 FILLER          PICTURE IS X VALUE SPACE.
  02 DEBUG-CONTENTS  PICTURE IS X(n).
```

The DEBUG-ITEM special register provides information about the conditions causing the running of a debugging section.

Before each debugging section is processed, DEBUG-ITEM is filled with spaces. The contents of the DEBUG-ITEM subfields are then updated according to the rules for the MOVE statement, with one exception: DEBUG-CONTENTS is updated as if the move were an alphanumeric-to-alphanumeric elementary move without conversion of data from one form of internal representation to another. After updating, each field contains:

- **DEBUG-LINE**: The compiler-generated statement number, right justified and padded on the left with zeros. For example, 000112.
- **DEBUG-NAME**: The first 30 characters of the name causing the debugging section to run. All qualifiers are separated by the word OF (subscripts or indexes are not entered in DEBUG-NAME).
- **DEBUG-SUB-1, DEBUG-SUB-2, DEBUG-SUB-3**: If the DEBUG-NAME is subscripted or indexed, the occurrence number of each level is entered in the respective DEBUG-SUB-n. If the item is not subscripted or indexed, these fields remain spaces.
- **DEBUG-CONTENTS**: Data is moved into DEBUG-CONTENTS as shown in Table 8. DEBUG-CONTENTS is the same size as the largest identifier in the program.
<table>
<thead>
<tr>
<th>Item Causing Debug Section To Run</th>
<th>DEBUG-LINE Contains Number of COBOL Statement Referring to</th>
<th>DEBUG-NAME Contains</th>
<th>DEBUG-CONTENTS Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier-n</td>
<td>identifier-n</td>
<td>identifier-n</td>
<td>Contents of identifier-n when control passes to debug section.</td>
</tr>
<tr>
<td>file-name-n</td>
<td>file-name-n</td>
<td>file-name-n</td>
<td>For READ: contents of record retrieved. Other references: spaces.</td>
</tr>
<tr>
<td>procedure-name-n ALTER reference</td>
<td>ALTER statement</td>
<td>procedure-name-n</td>
<td>procedure-name-n in TO PROCEED TO phrase</td>
</tr>
<tr>
<td>GO TO procedure-name-n</td>
<td>GO TO statement</td>
<td>procedure-name-n</td>
<td></td>
</tr>
<tr>
<td>procedure-name-n in SORT/MERGE INPUT/OUTPUT PROCEDURE</td>
<td>SORT/MERGE statement</td>
<td>procedure-name-n</td>
<td>“SORT INPUT” “SORT OUTPUT” “MERGE OUTPUT” as applicable</td>
</tr>
<tr>
<td>PERFORM statement transfer of control</td>
<td>This PERFORM statement</td>
<td>procedure-name-n</td>
<td>“PERFORM LOOP”</td>
</tr>
<tr>
<td>procedure-name-n in a USE procedure</td>
<td>Statement causing USE procedure running</td>
<td>procedure-name-n</td>
<td>“USE PROCEDURE”</td>
</tr>
<tr>
<td>Implicit transfer from previous sequential procedure</td>
<td>Previous statement processed in previous sequential procedure (see note)</td>
<td>procedure-name-n</td>
<td>“FALL THROUGH”</td>
</tr>
<tr>
<td>First entry into first non-declarative procedure</td>
<td>Line number of first statement in the procedure</td>
<td>First non-declarative procedure name</td>
<td>“START PROGRAM”</td>
</tr>
</tbody>
</table>

**Note:** If this paragraph is preceded by a section header and control is passed through the section header, the statement number refers to the section header.
Debugging Lines

Debugging lines can help determine the cause of an error. A debugging line is any line in a source program with a D coded in column 7 (the continuation area). If a debugging line contains nothing but spaces in Area A and Area B, it is considered a blank line.

Each debugging line must be written so that a syntactically correct program results whether the debugging lines are compiled into the program or syntax-checked, but are treated as documentation.

Successive debugging lines are permitted. Debugging lines can be continued. However, each continuation line must contain a D in column 7, and character-strings must not be broken across two lines.

Debugging lines can be specified only after the OBJECT-COMPUTER paragraph.

When the WITH DEBUGGING MODE clause is specified in the SOURCE-COMPUTER paragraph, all debugging lines are compiled as part of the object program.

When the WITH DEBUGGING MODE clause is omitted, all debugging lines are syntax-checked, but are treated as documentation.
Appendix C. Level of Language Support

ANSI X3.23-1985 COBOL Standard

The ANSI X3.23-1985 COBOL standard consists of eleven functional processing modules, seven of which are required and four of which are optional.

The seven required modules are: Nucleus, Sequential I-O, Relative I-O, Indexed I-O, Inter-Program Communication, Sort-Merge, and Source Text Manipulation. The four optional modules are: Report Writer, Communication, Debug and Segmentation.

Language elements within the modules may be classified as level 1 elements and level 2 elements. Elements within nine of the modules are divided into level 1 elements and level 2 elements. Two of the modules (SORT-MERGE and REPORT WRITER) contain only level 1 elements. For instance, Nucleus level 1 elements perform basic internal operations. Nucleus level 2 elements provide for more extensive and sophisticated internal processing.

The three subsets of Standard COBOL are the high subset, the intermediate subset, and the minimum subset. Each subset is composed of a level of the seven required modules: Nucleus, Sequential I-O, Relative I-O, Indexed I-O, Inter-Program Communication, Sort-Merge, and Source Text Manipulation. The four optional modules (Report Writer, Communication, Debug and Segmentation) are not required in the three subsets of Standard COBOL.

The high subset is composed of all language elements of the highest level of all required modules. That is:

- Level 2 elements from Nucleus, Sequential I-O, Relative I-O, Indexed I-O, Inter-Program Communication, and Source Text Manipulation
- Level 1 elements from Sort-Merge.

The intermediate subset is composed of all language elements of level 1 of all required modules. That is:

- Level 1 elements from Nucleus, Sequential I-O, Relative I-O, Indexed I-O, Inter-Program Communication, Sort-Merge, and Source Text Manipulation.

The minimum subset is composed of all language elements of level 1 of the Nucleus, Sequential I-O, and Inter-Program Communication modules.

The four optional modules are not an integral part of any of the subsets. However, none, all, or any combination of the optional modules may be associated with any of the subsets.

COBOL/400 Level of Language Support

The COBOL/400 compiler supports:

- Level 1 of the Nucleus, Sequential I-O, Relative I-O, Indexed I-O, Inter-Program Communication, Sort-Merge, and Source Text Manipulation modules
- Level 2 of the Debug and Segmentation modules.
The Report Writer and Communication modules of ANSI X3.23-1985 COBOL are not supported by the COBOL/400 compiler.

The level of support provided by the COBOL/400 compiler is represented in the table below. The table:
- Shows the level of COBOL/400 compiler support for each functional processing module of the ANSI X3.23-1985 COBOL standard
- Describes each module.

Following is an explanation of the notation used within the table:

A 3-character code that identifies the module. In this example, the Segmentation module, is referenced.

The level of this module supported by the COBOL/400 compiler. In this example, support is provided for the higher of the two levels of the Segmentation module.

The range of levels of support defined by the ANSI X3.23-1985 COBOL standard. A level of 0 means a minimum standard COBOL does not need to support this module to conform to the standard.

<table>
<thead>
<tr>
<th>COBOL/400 Level of Language Supported</th>
<th>Module Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nucleus 1 NUC 1,2</td>
<td>Contains the language elements necessary for internal processing of data within the four basic divisions of a program and the capability for defining and accessing tables.</td>
</tr>
<tr>
<td>Sequential I-O 1 SEQ 1,2</td>
<td>Provides access to file records by the established sequence in which they were written to the file.</td>
</tr>
<tr>
<td>Relative I-O 1 REL 0,2</td>
<td>Provides access to records in either a random or sequential manner. Each record is uniquely identified by an integer that represents the record’s logical position in the file.</td>
</tr>
<tr>
<td>Indexed I-O 1 INX 0,2</td>
<td>Provides access to records in either random or sequential manner. Each record in an indexed file is uniquely identified by a record key.</td>
</tr>
<tr>
<td>Inter-program Communication 1 IPC 1,2</td>
<td>Allows a COBOL program to communicate with other programs through transfers of control and access to common data items.</td>
</tr>
<tr>
<td>Sort-Merge 1 SRT 0,1</td>
<td>Orders one or more files of records, or combines two or more identically ordered files according to user-specified keys.</td>
</tr>
<tr>
<td>Source-Text Manipulation 1 STM 0,2</td>
<td>Allows insertion of predefined COBOL text into a program at compile time.</td>
</tr>
</tbody>
</table>
### Table 9 (Page 2 of 2). Level of COBOL/400 Compiler Support

<table>
<thead>
<tr>
<th>COBOL/400 Level of Language Supported</th>
<th>Module Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report Writer 0 RPW 0,1</td>
<td>Provides semiautomatic production of printed reports.</td>
</tr>
<tr>
<td>Communications 0 COM 0,2</td>
<td>Provides the ability to access, process, and create messages or portions of messages; also allows communication through a Message Control System with local and remote communication devices.</td>
</tr>
<tr>
<td>Debug 2 DEB 0,2</td>
<td>Allows you to specify statements and procedures for debugging.</td>
</tr>
<tr>
<td>Segmentation 2 SEG 0,2</td>
<td>Provides the overlaying at object time of Procedure Division sections.</td>
</tr>
</tbody>
</table>

**SAA Common Programming Interface (CPI) Support**

Source file QILBINC in product libraries QLBL and QLBLP contains members that hold specifications for multiple SAA Common Programming Interfaces. These specifications describe parameter interfaces. This file is IBM-owned and should not be changed.

If you want to customize any of the specifications, you must copy any members that you want to change to a source file in one of your libraries. You can use the Copy File (CPYF) command to do this. For more information about the CPYF command, refer to the CL Reference.

If you copy these specifications to your library, you must refresh your copies when a new product release is installed, or when any changes are made using a Program Temporary Fix (PTF). IBM provides maintenance for these specifications only in the libraries in which they are distributed.
Appendix D. COBOL/400 Messages, the FIPS Flagger, and SAA Flagging

COBOL/400 Messages

This appendix provides a general description of messages that IBM supplies with the COBOL/400 licensed program.

Interactive Messages

In an interactive environment, messages are displayed on the work station display. They can appear on the current display as a result of the running of the program or in response to your keyed input to prompts, menus, command entry displays, or Application Development Tools (Appl Dev Tools). The messages can also appear on request, as a result of a display command or an option on a menu.

The messages for the COBOL/400 licensed program begin with an LSC, LBE, or LBL prefix.

The LSC messages are issued by the COBOL/400 syntax checker when the Source Entry Utility (SEU) is used to enter your COBOL/400 source. For example, you see the following display after incorrectly entering the program name in the PROGRAM-ID paragraph.

```
           Columns . . . : 1 71       Edit       XMPLIB/QLBSRC
SEU==>                        -------------------------------
FMT CB ........A+++B+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                        *************** Beginning of data ***************
0000.10 IDENTIFICATION DIVISION.
0000.20 PROGRAM-ID. #TESTPR.
0000.70 ENVIRONMENT DIVISION.
0000.90 SOURCE-COMPUTER. IBM-AS400.
                        *************** End of data ***************
```

F3=Exit  F4=Prompt  F5=Refresh  F9=Retrieve  F10=Cursor
F16=Repeat find  F17=Repeat change  F24=More keys
# not in COBOL character set. Line rejected.

Figure 109. Example of a COBOL/400 Syntax Checker Message
LBE messages provide you with additional information about system operation during run time. For example, you might see the following display if you have a run-time error:

![Display Program Messages](image)

Figure 110. Run-Time Error Message

If you move the cursor to the line on which message number CPF4101 is indicated and press either the HELP key or F1, the LBE message information is displayed as shown:

![Additional Message Information](image)

Figure 111. Run-Time Error Message—Second-Level Text

LBE messages 7900 to 7999 are used as headings for information printed during a COBOL/400 formatted dump.
The LBL messages are described under “Compilation Messages” below.

“Responding to Messages” on page 329 explains how to display second-level message text and how to reply to messages.

Compilation Messages
LBL messages are printed in the program listing when errors are found during program compilation. The LBL messages include the message issued when Federal Information Processing Standard (FIPS) flagging is requested; for more information on the FIPS messages, refer to page 331 in this appendix.

Program Listings
In the compiler output, the COBOL/400 messages listing follows the source listing. The COBOL/400 messages listing gives the message identifier, severity, text, usually the location of the error, and the messages summary.

For more information about Program Listings, see “Source Listing” on page 41.

Responding to Messages
In an interactive environment, a message is indicated by one or several of these conditions:

• A brief message (called first-level text) on the message line
• Reverse image highlighting of the input field in error
• A locked keyboard
• The sound of an alarm (if the alarm option is installed).

The following paragraphs briefly describe some methods of responding to error messages; more information is available in the New User’s Guide and the Application Development Tools publications.

If the necessary correction is obvious from the initial display, you can press the Error Reset key (if the keyboard is locked), enter the correct information, and continue your work.

If the message requires that you choose a reply (such as C to cancel, D to dump COBOL identifiers, F to dump all variables, or G to resume processing at the next COBOL statement), the reply options are shown in parentheses in the first-level message text. For an example, see Figure 110 on page 328.

If the information on the initial information display does not provide sufficient data for you to handle the error, you can press the HELP key (after positioning the cursor to the message line, if required) to get a second-level display with additional information about how to correct this error. To return to the initial display, press the Enter key; then press the Error Reset key (if the keyboard is locked), and make your correction or response.

If the error occurs when you are compiling or running a program, you might need to modify your COBOL/400 source statements or control language (CL) commands. Refer to the SEU User’s Guide and Reference for information on how to change the statements.
COBOL Message Descriptions

The messages for the COBOL/400 licensed program begin with prefixes LSC, LBE, or LBL.

The LSC messages are issued by the COBOL syntax checker when SEU is used to enter your COBOL source.

The LBE messages provide you with additional information about system operation during run time.

The LBL messages are compiler-generated messages.

Message numbers are assigned as follows:

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBE7000 through LBE7199</td>
<td>Escape Messages</td>
</tr>
<tr>
<td>LBE7200 through LBE7999</td>
<td>Run-time messages</td>
</tr>
<tr>
<td>LBE9001</td>
<td>Escape message</td>
</tr>
<tr>
<td>LBL0000 through LBL0999</td>
<td>Messages with severity less than 30</td>
</tr>
<tr>
<td>LBL1000 through LBL1999</td>
<td>Messages with severity greater than or equal to 30</td>
</tr>
<tr>
<td>LBL8000 through LBL8799</td>
<td>FIPS Flagger messages</td>
</tr>
<tr>
<td>LBL8800 through LBL8899</td>
<td>SAA Flagging messages</td>
</tr>
<tr>
<td>LSC0000 through LSC1999</td>
<td>Syntax checker messages</td>
</tr>
</tbody>
</table>

Severity Levels

The COBOL/400 licensed program provides the following message severity levels:

<table>
<thead>
<tr>
<th>Severity</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Informational: This level is used to convey information to the user.</td>
</tr>
<tr>
<td></td>
<td>No error has occurred. Informational messages are listed only when the FLAG (00) option is specified.</td>
</tr>
<tr>
<td>10</td>
<td>Warning: This level indicates that an error was detected but is not serious enough to interfere with the running of the program.</td>
</tr>
<tr>
<td>20</td>
<td>Error: This level indicates that an error was made, but the compiler is taking a recovery that might yield the desired code.</td>
</tr>
<tr>
<td>30</td>
<td>Severe Error: This level indicates that a serious error was detected.</td>
</tr>
<tr>
<td></td>
<td>Compilation is completed, but running of the program cannot be attempted.</td>
</tr>
<tr>
<td>40</td>
<td>Unrecoverable: This level usually indicates a user error that forces termination of processing.</td>
</tr>
<tr>
<td>50</td>
<td>Unrecoverable: This level usually indicates a compiler error that forces termination of processing.</td>
</tr>
<tr>
<td>99</td>
<td>Action: Some manual action is required, such as entering a reply, changing printer forms, or replacing diskettes.</td>
</tr>
</tbody>
</table>

Note: 00, 10, and 20 messages are suppressed when the FLAG(30) option of the PROCESS statement is used or the CRTCBLPGM command specifies FLAG(30) and is not overridden by the PROCESS statement. See “Using the PROCESS Statement to Specify Compiler Options” on page 32 for further information.
The compiler always attempts to provide full diagnostics of all source text in the program, even when errors have been detected. If the compiler cannot continue on a given statement, the message states that the compiler cannot continue and that it will ignore the rest of the statement. When this error occurs, the programmer should examine the entire statement.

The OS/400 message facility is used to produce all messages. The COBOL/400 compiler messages reside in the message file, QLBLMSG, and the run-time messages reside in the message file, QLBLMSGE.

Substitution variables and valid reply values are determined by the program sending the message, not by the message description stored in the message file. However, certain elements of a message description can be changed: for example, the text, severity level, default response, or dump list. To effect such changes, you need to define another message description using an Add Message Description (ADDMSGD) command, place the modified description in a user-created message file, and specify that file in the Override Message File (OVRMSGF) command. Using the OVRMSGF command allows the compiler to retrieve messages from the specified file. See the ADDMSGD and OVRMSGF commands in the CL Reference for additional information.

**CAUTION:** Overriding an IBM-supplied message with a user-created message can produce results you do not anticipate. If reply values are not retained, the program might not respond to any replies. Changing default replies on *NOTIFY type messages could affect the ability of the program to run in unattended mode. Changing the severity could cancel a job not previously canceled. Be cautious when overriding IBM-supplied messages with user-created messages.

---

**The Federal Information Processing Standard (FIPS) Flagger**

The FIPS flagger can be specified to monitor a FIPS COBOL subset, any of the optional modules, all of the obsolete language elements, or a combination of a FIPS COBOL subset, optional modules and all obsolete elements.

The monitoring is an analysis that compares the syntax used in the source program with the syntax included in the user-selected FIPS subset and optional modules. Any syntax used in the source program that does not conform to the selected FIPS COBOL subset and optional modules is identified. Any syntax for an obsolete language element used in the source program will also be identified (depending on the compiler option chosen). See page 25 for more information on the parameters for FIPS flagging.

1986 FIPS COBOL specifications are the language specifications contained in ANSI X3.23-1985 COBOL. FIPS COBOL is subdivided into three subsets and four optional modules. The three subsets are identified as Minimum, Intermediate and High. The four optional modules are Report Writer, Communication, Debug, and Segmentation. These four optional modules are not an integral part of any of the subsets; however, none, all, or any combination of the optional modules may be associated with any of the subsets. Any program written to conform to the 1986 FIPS standard must conform to one of the subsets of 1986 FIPS COBOL.

---

1 If an IBM-supplied message must be changed and replaced in its message file, call your service representative.
Table 10 on page 332 shows the 1985 ANSI Standard COBOL processing modules included in each of the subsets of 1986 FIPS COBOL.

Following is an explanation of the notation used within the table:

A 3-character code that identifies the module. In this example, the Segmentation module, is referenced.

The level of this module supported by the 1986 FIPS COBOL standard. In this example, support is provided for the higher of the two levels of the Segmentation module.

The range of levels of support defined by the ANSI X3.23-1985 COBOL standard. A level of 0 means a minimum standard COBOL does not need to support this module to conform to the standard.

### Table 10. 1985 American National Standard COBOL and 1986 FIPS Levels

<table>
<thead>
<tr>
<th>1985 ANSI Module Name</th>
<th>High FIPS</th>
<th>Intermediate FIPS</th>
<th>Minimum FIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nucleus</td>
<td>2 NUC 1,2</td>
<td>1 NUC 1,2</td>
<td>1 NUC 1,2</td>
</tr>
<tr>
<td>Sequential I-O</td>
<td>2 SEQ 1,2</td>
<td>1 SEQ 1,2</td>
<td>1 SEQ 1,2</td>
</tr>
<tr>
<td>Relative I-O</td>
<td>2 REL 0,2</td>
<td>1 REL 0,2</td>
<td>0 REL 0,2</td>
</tr>
<tr>
<td>Indexed I-O</td>
<td>2 INX 0,2</td>
<td>1 INX 0,2</td>
<td>0 INX 0,2</td>
</tr>
<tr>
<td>Source-Text Manipulation</td>
<td>2 STM 0,2</td>
<td>1 STM 0,2</td>
<td>0 STM 0,2</td>
</tr>
<tr>
<td>Sort-Merge</td>
<td>1 SRT 0,1</td>
<td>1 SRT 0,1</td>
<td>0 SRT 0,1</td>
</tr>
<tr>
<td>Inter-Program Communication</td>
<td>2 IPC 1,2</td>
<td>1 IPC 1,2</td>
<td>1 IPC 1,2</td>
</tr>
<tr>
<td>Report Writer</td>
<td>0, or 1 RPW 0,1</td>
<td>0, or 1 RPW 0,1</td>
<td>0, or 1 RPW 0,1</td>
</tr>
<tr>
<td>Segmentation</td>
<td>0,1 or 2 SEG 0,2</td>
<td>0,1 or 2 SEG 0,2</td>
<td>0,1 or 2 SEG 0,2</td>
</tr>
<tr>
<td>Debug</td>
<td>0,1 or 2 DEB 0,2</td>
<td>0,1 or 2 DEB 0,2</td>
<td>0,1 or 2 DEB 0,2</td>
</tr>
<tr>
<td>Communications</td>
<td>0,1 or 2 COM 0,2</td>
<td>0,1 or 2 COM 0,2</td>
<td>0,1 or 2 COM 0,2</td>
</tr>
</tbody>
</table>

**Note:** The COBOL/400 compiler supports the Segmentation and Debug optional modules.

Elements that are specified in the COBOL/400 source program and that are not included in 1986 FIPS COBOL are flagged as described in Appendix C, “Level of Language Support” on page 323.
SAA Flagging

You can choose to perform SAA flagging to determine if the COBOL/400 functions that you are using are portable to other SAA COBOL environments.

Flagging is performed on those COBOL/400 functions that are outside of SAA COBOL, such as:
- COBOL/400 extensions
- COBOL/400 compiler limits
- Non-SAA reserved words
- Compiler options.

In this way, you can write programs that conform to the SAA COBOL definition.

For an example of SAA flagging in a compiler listing, see Figure 12 on page 47.
To perform SAA flagging through the CRTCBLPGM CL command, specify SAAFLAG(*FLAG). To perform SAA flagging through a PROCESS statement, specify SAAFLAG.

To compile a program to conform to the SAA definition, using the CRTCBLPGM command, specify the following:
- OPTION(*QUOTE *NOSEQUENCE *NONUMBER)
- GENOPT(*CRTC *DUPKEYCHK *SYNC)
- SAAFLAG(*FLAG)

If you use the PROCESS statement, specify the following:
- QUOTE, NOSEQUENCE, NONUMBER, CRTF,
  DUPKEYCHK, SYNC, SAAFLAG.

For more information about specifying the option for SAA flagging, see the SAAFLAG parameter on page 25, and the “Using the PROCESS Statement to Specify Compiler Options” on page 32.

For information about compiler limits, see the Compiler Limits appendix in the COBOL/400 Reference.
Appendix E. Differences Between ANSI 74 COBOL and ANSI 85 COBOL

This appendix identifies the ANSI 85 COBOL language elements that are incompatible with ANSI 74 COBOL. These items identify the changes and conditions that ANSI 74 COBOL users need to be aware of when migrating to ANSI 85 COBOL.

See “Industry Standards Used in Compiler Design” on page xiii for more information on ANSI 85 COBOL.

Migrating ANSI 74 COBOL Programs to ANSI 85 COBOL

The following are some of the new features or changes to ANSI 85 COBOL that could affect ANSI 74 COBOL programs:

- The keyword ALPHABET must precede alphabet-name within the alphabet-name clause of the SPECIAL-NAMES paragraph. An alphabet-name is a user-defined word in the SPECIAL-NAMES paragraph that names a character set or collating sequence.
- The relative key data item specified in the RELATIVE KEY phrase must not contain the PICTURE symbol "P."
- The ALPHABETIC class test is true for uppercase letters, lowercase letters, and the space character.
- When there is no next statement to be processed in a called program, an implicit EXIT PROGRAM is run.
- No two files in a MERGE statement can be specified in the SAME AREA or SAME SORT-MERGE AREA clause. The only files in a MERGE statement that can be specified in the SAME RECORD AREA clause are those associated with the GIVING phrase.
- Within the READ statement, the INTO phrase cannot be specified unless:
   - All records associated with the file and the data item specified in the INTO phrase are group items or elementary alphanumeric items, or only one record description is subordinate to the file description entry.
- Within the RETURN statement, the INTO phrase cannot be specified unless:
   - All records associated with the file and data item specified in the INTO phrase are group items or elementary alphanumeric items, or only one record description is subordinate to the sort-merge file description entry.
- File position indicator - the concept of a current record pointer has been changed to a file position indicator.
- Reserved words - new reserved words have been added.
- I/O status - new I/O status values have been added.
- Pseudo-text-1 on the COPY statement must not consist entirely of a separator comma or a separator semicolon.
- A data item appearing in the USING phrase of the Procedure Division header must not have a REDEFINES clause in its data description entry.
• If the FOOTING phrase is not specified, no end-of-page condition independent of the page overflow condition exists.

• The NO REWIND phrase cannot be specified in a CLOSE statement having the REEL/UNIT phrase.

• The CANCEL and STOP RUN statements close all open files.

• When a receiving item is a variable-length data item and contains the object of the DEPENDING ON phrase, the maximum length of the item will be used.

• Within the VARYING ... AFTER phrase of the PERFORM statement, identifier-2 is augmented before identifier-5 is set.

• Any subscripts for identifier-4 in the DIVIDE statement REMAINDER phrase are evaluated after the result of the DIVIDE operation is stored in identifier-3 of the GIVING phrase.

• The phrase ADVANCING PAGE and END-OF-PAGE must not both be in a single WRITE statement.

• The picture character-string of an alphabetic item can contain only the symbol “A.” No editing is allowed for the alphabetic data category.

  Note: An alphabetic character is a letter or a space character.

• When a data item described by a PICTURE containing the character “P” is referenced, the digit positions specified by “P” are considered to contain zeros in the following operations:
  – Any operation requiring a numeric sending operand
  – A MOVE statement where the sending operand is numeric and its PICTURE character-string contains the symbol “P”
  – A MOVE statement where the sending operand is numeric edited and its PICTURE character-string contains the symbol “P” and the receiving operand is numeric or numeric edited
  – A comparison operation where both operands are numeric.

• The literal in the CURRENCY SIGN clause cannot be a figurative constant.

• If the COPY statement appears in a comment-entry, it is considered part of the comment-entry.

• The following special cases of exponentiation are defined:
  – If an expression having a zero value is raised to a negative or zero power, the size error condition exists.
  – If the evaluation of the exponentiation yields both a positive and a negative real number, the positive number is returned.
  – If no real number exists as the result of the evaluation, the size error condition exists.

• When the figurative constant ALL literal is not associated with another data item, the length of the string is the length of the literal.
Appendix F. Supporting International Languages with Double-Byte Character Sets

IBM Extension

This appendix describes only those enhancements made to the COBOL programming language for writing programs that process double-byte characters.

Specifically, this appendix describes where you can use Double-Byte Character Set (DBCS) characters in each portion of a COBOL program, and considerations for working with DBCS data in the COBOL/400 language.

There are two ways to specify DBCS characters:

- Bracketed-DBCS
- DBCS-graphic data

In general, COBOL handles bracketed-DBCS characters in the same way it handles alphanumeric characters. Bracketed-DBCS is a character string in which each character is represented by two bytes. The character starts with a shift-out (SO) character, and ends with a shift-in (SI) character. It is up to you to know (or have the COBOL program check) which data items contain DBCS characters, and to make sure the program receives and processes this information correctly.

You can now use DDS descriptions that define DBCS-graphic data fields with your COBOL/400 programs. DBCS-graphic pertains to a character string where each character is represented by two bytes. The character string does not contain shift-out or shift-in characters. You cannot use source programs containing graphic data. For information on specifying graphic data items with your COBOL/400 programs, refer to “DBCS-Graphic Fields” on page 133.

Using DBCS Characters in Literals

Types of Literals
There are two types of literals in which you can use DBCS characters: the DBCS literal and the mixed literal. A mixed literal consists of Double-Byte Character Set (DBCS) and Single-Byte Character Set (SBCS) characters.

DBCS Literals: The COBOL compiler recognizes DBCS characters in DBCS literals when you use the GRAPHIC option on the PROCESS statement.

Note: The GRAPHIC option on the PROCESS statement is not to be confused with the *GRAPHIC value in the CVTOPT parameter of the CRTCBLPGM command and the CVTGRAPHIC option on the PROCESS statement, which are used to specify double-byte graphic data from a DDS description. For more information on specifying graphic data, refer to “DBCS-Graphic Fields” on page 133.

DBCS/SBCS Literals: The COBOL compiler recognizes DBCS characters in DBCS/SBCS (mixed) literals, when you are on a DBCS system and the GRAPHIC option on the PROCESS statement is not specified.
How to Specify Literals Containing DBCS Characters

When you specify any literal that contains DBCS characters, follow the same rules that apply in specifying alphanumeric literals, as well as the following rules specific to the literal types:

**How to Specify a DBCS Literal:** When you specify a DBCS literal, keep in mind the following:

The format for a DBCS literal is:

"0EKiK20F"

- A quotation mark opens and closes the literal.
- A shift-out character (0E) immediately follows the initial quotation mark and occupies 1 byte. A **shift-out** character is a control character (hex 0E) that indicates the start of a string of double-byte characters.
- A shift-in character (0F) immediately precedes the final quotation mark and occupies 1 byte. A **shift-in** character is a control character (hex 0F) that indicates the end of a string of double-byte characters.
- All DBCS characters appear between the shift-out and shift-in characters.
- Only DBCS characters may appear in the literal (null strings are valid).

The maximum length of a DBCS literal is 80 DBCS characters, including the shift control characters. (These counted together are equivalent in length to one DBCS character.) The shift control characters are part of the literal, and take part in all operations.

See “How to Continue DBCS Literals on a New Line” on page 339 for information on how to extend DBCS literals.

**How to Specify a DBCS/SBCS Literal:** When you specify a DBCS/SBCS literal, keep in mind the following:

- DBCS/SBCS literals can take many different forms. The following is only one possible example:
  
  "SINGLE0EKiK20FBYTES"

- USAGE DISPLAY must be either explicit or implicit.
- A quotation mark opens and closes the literal.
- EBCDIC characters can appear before or after any DBCS string in the mixed literal.
- All DBCS strings appear between shift-out and shift-in characters.
- Double all SBCS quotation marks that occur within the literal. DBCS quotation marks within the literal do not require doubling.
- You can use null DBCS strings (shift-out and shift-in characters without any DBCS characters) only when the literal contains at least one SBCS character.

The shift-out and shift-in characters cannot be nested.

The shift control characters are part of the literal, and take part in all operations.
DBCS/SBCS literals cannot continue across lines. They are restricted to the space of AREA B on one line.

Other Considerations

Quotation Marks: Although the preceding discussion uses the term a quotation mark to describe the character that identifies a literal, the character actually used can vary depending upon the option specified on the CRTCLBLPGM CL command, or on the PROCESS statement. If you specify the APOST option, an apostrophe (') is used. Otherwise, a quotation mark (") is used. In this appendix, a quotation mark refers to both an apostrophe and a quotation mark. The character that you choose does not affect the rules for specifying a literal.

Shift Characters: The shift-out and shift-in characters separate EBCDIC characters from DBCS characters. They are part of both the DBCS and the DBCS/SBCS literal. Therefore, the shift code characters participate in all operations when they appear in either DBCS or DBCS/SBCS literals.

How the COBOL Compiler Checks DBCS Characters

When the COBOL compiler finds a DBCS string, it checks the DBCS string by scanning it one DBCS character at a time.

The following conditions cause the COBOL compiler to diagnose a literal containing DBCS characters as not valid:

- The syntax for the literal is incorrect.
- The DBCS literal is longer than one line and does not follow the rules for continuing nonnumeric literals. (See “How to Continue DBCS Literals on a New Line” for more information.)
- The DBCS/SBCS literal is longer than one line.

When the COBOL compiler finds a DBCS literal that is not valid, it generates an error message, and then processes the literal as an alphanumeric literal.

For each DBCS or SBCS literal that is not valid, the compiler generates an error message and accepts or ignores the literal.

How to Continue DBCS Literals on a New Line

To continue a DBCS literal onto another line of source code, do all of the following:

- Place a shift-in character in either column 71 or column 72 of the line to be continued (If you put the shift-in character in column 71, the blank in column 72 is ignored)
- Place a hyphen (-) in column 7 (the continuation area) of the new line
- Place a quotation mark, then a shift-out character, and then the rest of the literal in Area B of the new line.

For example:

```
-A 1 B
...
01 DBCS1 PIC X(12) VALUE "0E1K2K30F"...
- "0E4K50F".
...
```

Appendix F. Supporting International Languages with Double-Byte Character Sets
The value of DBCS1 is "$E_{0\text{K1K2K3K4K5}_F}$."

The shift-in character, quotation mark, and shift-out character used to continue a line are not counted in the length of the DBCS literal. The first shift-out and final shift-in characters are counted.

Where You Can Use DBCS Characters in a COBOL Program
In general, you can use DBCS, or DBCS/SBCS literals wherever nonnumeric literals are allowed. Literals for the following, however, cannot include double-byte characters:

- ALPHABET-NAME clause
- CURRENCY SIGN clause
- ASSIGN clause
- CLASS clause
- CALL statement
- CANCEL statement.

Note: You cannot use DBCS characters for COBOL words or names. See the COBOL/400 Reference for information on rules for formatting COBOL system-names, reserved words, and user-defined words such as data names and file names.

How to Write Comments
You can write comments containing DBCS characters in a COBOL program by putting an asterisk (*) or slash (/) in column seven of the program line. Either symbol causes the compiler to treat any information following column seven as documentation. The slash also causes a page eject. Because the COBOL compiler does not check the contents of comment lines, DBCS characters in comments are not detected. DBCS characters that are not valid can cause the compiler listing to print improperly.

Identification Division
You can put comment entries that contain DBCS characters in any portion of the Identification Division except the PROGRAM-ID paragraph. The program name specified in the PROGRAM-ID paragraph must be alphanumeric.

Environment Division

Configuration Section
You can use DBCS characters in comment entries only in the Configuration Section paragraph. All function-names, mnemonic-names, condition-names, and alphabet-names must be specified with alphanumeric characters. For the SOURCE-COMPUTER and the OBJECT-COMPUTER entry, use the alphanumeric computer name:

IBM-AS400

You cannot use DBCS or DBCS/SBCS literals in the Configuration Section. Instead, use alphanumeric literals to define an alphabet-name and the literal in the CURRENCY SIGN clause of the SPECIAL-NAMES paragraph. There is no DBCS alphabet. Use the EBCDIC character set instead.
Input-Output Section
Specify all data names, file names, and assignment names using alphanumeric characters. You can use DBCS characters in comments.

For indexed files, the data name in the RECORD KEY clause can refer to a DBCS or DBCS/SBCS data item within a record. The number of fields in the record, plus the number of positions occupied by the record key, together cannot be greater than 120.

Note: Each DBCS character occupies two positions, and the shift control characters each occupy one position. Ensure that both the data description of the key and the key position within the file match those specified when you created the file.

You cannot use DBCS and DBCS/SBCS data as the RELATIVE KEY in relative files.

File Control Paragraph
ASSIGN Clause: You cannot use literals containing DBCS characters in the ASSIGN clause to specify an external medium such as a printer or a database.

Data Division

File Section
For the FD (File Description) Entry, you can use DBCS or DBCS/SBCS data items or literals in the VALUE OF clause. The DATA RECORDS clause can refer to data items only. Because the COBOL/400 compiler treats both the VALUE OF clause and the DATA RECORDS clause in the File Section as documentation, neither clause has any effect when you run the program. However, the COBOL compiler checks all literals in the VALUE OF clause to make sure they are valid.

For magnetic tapes, the system can only read DBCS characters from, or write DBCS characters to, the tape in the EBCDIC format. The system cannot perform tape functions involving a tape in the ASCII format. Define the alphabet-name in the CODE-SET clause as NATIVE. Use alphanumeric characters to specify the alphabet-name.

Working-Storage Section
REDEFINES Clause: The existing rules for redefining data also apply to data that contains DBCS characters. When you determine the length of a redefining or redefined data item, remember that each DBCS character is twice as long as an alphanumeric character.

Also, ensure that redefined data items contain the shift control characters when and where necessary.

OCCURS Clause: Use this clause to define tables for storing DBCS or DBCS/SBCS data. If you specify the ASCENDING/DESCENDING KEY phrase, COBOL assumes the contents of the table are in the EBCDIC program collating sequence. The shift control characters in DBCS and DBCS/SBCS data take part in the collating sequence.

For more information about handling tables that contain DBCS characters, see “Table Handling–SEARCH Statement” on page 348.
**JUSTIFIED RIGHT Clause:** Use the JUSTIFIED RIGHT clause to align DBCS or DBCS/SBCS data at the rightmost position of an elementary receiving field. If the receiving field is shorter than the sending field, COBOL truncates the rightmost characters. If the receiving field is longer than the sending field, COBOL pads (fills) the unused space on the left of the receiving field with blanks.

The JUSTIFIED clause does not affect the initial setting in the VALUE clause.

**VALUE Clause:** You can use DBCS or DBCS/SBCS literals to specify an initial value for a data item that is not numeric, or to define values for level-88 condition-name entries.

Any shift control characters in the literal are considered part of the literal’s picture string, except when used to continue a new line. When you continue a DBCS literal, the compiler does not include the shift-in character in column 71 or 72, or the initial quotation mark (") and shift-out character on the continued line as part of the DBCS literal. Make certain, however, that the DBCS literal does not exceed the size of the data item specified in the PICTURE clause, otherwise truncation occurs.

**Note:** DBCS/SBCS mixed literals cannot be continued to a new line.

When you use literals that contain DBCS characters in the VALUE clause for level-88 condition-name entries, COBOL treats the DBCS characters as alphanumeric. Therefore, follow the rules for specifying alphanumeric data, including allowing a THROUGH option. This option uses the normal EBCDIC collating sequence, but remember that shift control characters in DBCS and DBCS/SBCS data take part in the collating sequence.

**PICTURE Clause:** Use the PICTURE symbol X to define DBCS and DBCS/SBCS data items. Because DBCS characters are twice as long as alphanumeric, and are enclosed within shift control characters, you would define a DBCS data item containing n DBCS characters as

PICTURE X(2n+2)

A DBCS/SBCS data item containing m SBCS characters, and one string of n DBCS characters would be defined as

PICTURE X(m+2n+2)

You can use all edited alphanumeric PICTURE symbols for DBCS and DBCS/SBCS data items. The editing symbols have the same effect on the DBCS data in these items as they do on alphanumeric data items. Check that you have obtained the desired results.

**RENAMES Clause:** Use this clause to specify alternative groupings of elementary data items. The existing rules for renaming alphanumeric data items also apply to DBCS and DBCS/SBCS data items.
**Procedure Division**

**Declaratives**
An identifier in the USE FOR DEBUGGING sentence of the DECLARATIVES section can refer to a DBCS or a DBCS/SBCS data item.

You cannot use DBCS characters for file names or procedure names in the USE FOR DEBUGGING sentence.

**Conditional Expressions**
Because condition-names (level-88 entries) can refer to data items that contain DBCS characters, you can use the condition-name condition to test this data. (See “VALUE Clause” on page 342.) Follow the rules listed in the COBOL/400 Reference for using conditional variables and condition-names.

You can use DBCS or DBCS/SBCS data items or literals as the operands in a relation condition. Because COBOL treats DBCS data as alphanumeric, all comparisons occur according to the rules for alphanumeric operands. Keep the following in mind:

- The system does not recognize the mixed content.
- The system uses the shift codes in comparisons of DBCS and DBCS/SBCS data.
- The system compares the data using either the EBCDIC collating sequence, or a user-defined sequence.
- In a comparison of DBCS or DBCS/SBCS items with similar items of unequal size, the smaller item is padded on the right with EBCDIC spaces.

See “SPECIAL-NAMES Paragraph” section in the COBOL/400 Reference for more information.

You can use class conditions and switch status conditions as described in the COBOL/400 Reference.

**Input/Output Statements**

**ACCEPT Statement:** The input data received from a device by using a Format 1 ACCEPT statement can include DBCS or DBCS/SBCS data. All DBCS and DBCS/SBCS data must be identified by the proper syntax. The input data, including shift control characters, replaces the existing contents of the identifier. COBOL does not perform editing or error checking on the data.

If you use the Format 3 ACCEPT statement to get OPEN-FEEDBACK information about a file, that information includes a field showing whether the file has DBCS or DBCS/SBCS data.

Information received from the local data area by a Format 4 ACCEPT statement can include DBCS or DBCS/SBCS character strings. Information received replaces the existing contents. COBOL does not perform any editing or checking for errors. This also applies to information received from the PIP data area by a Format 5 ACCEPT statement.

Using the Format 6 ACCEPT statement, you can get the attributes of a work station display and its keyboard. For display stations that can display DBCS characters,
the system sets the appropriate value in the ATTRIBUTE-DATA data item. You cannot use DBCS characters to name a device.

If you use an extended (Format 7) ACCEPT statement for field-level work station input, you must ensure that DBCS data is not split across lines. COBOL does not perform any editing or checking for errors.

**DISPLAY Statement:** You can specify DBCS or DBCS/SBCS data items or literals in the DISPLAY statement. You can mix the types of data. DBCS and DBCS/SBCS data, from either data items or literals, is sent as it appears to the program device or local data area that is the target named on the DISPLAY statement.

Because COBOL does not know the characteristics of the device on which data is being displayed, you must make sure that the DBCS and DBCS/SBCS data is correct. It may be necessary to specify the extended display option *NOUNDSPIC (or the equivalent process statement parameter option) when the program is compiled, to ensure that a workstation can handle DBCS data correctly.

**Note:** ALL is a valid option for mixed literals.

If you use an extended (Format 3) DISPLAY statement for field-level work station output, you must ensure that DBCS data is not split across lines.

**READ Statement:** You can use DBCS or DBCS/SBCS data items as the RECORD KEY for an indexed file. See “Input-Output Section” on page 341 for more information.

**INTO Phrase:** You can read a record into a DBCS or a DBCS/SBCS data item using the INTO phrase. This phrase causes a MOVE statement (without the CORRESPONDING option) to be performed. The compiler moves DBCS and DBCS/SBCS data in the same manner that it moves alphanumeric data. It does not make sure that this data is valid.

**REWRITE Statement:** Use the FROM phrase of this statement to transfer DBCS or DBCS/SBCS data from a DBCS or a DBCS/SBCS data item to an existing record. The FROM phrase causes both types of data to be moved in the same manner as the INTO phrase with the READ statement. (See “READ Statement.”)

**START Statement:** If you use DBCS characters in the key of an indexed file, specify a corresponding data item in the KEY phrase of the START statement.

One of the following must be true:

- The data item must be the same as the data item specified in the RECORD KEY clause of the FILE-CONTROL paragraph.
- The data item has the same first character as the record key and is not longer than the record key.

You can specify valid operators (such as EQUAL, GREATER THAN, NOT LESS THAN) in the KEY phrase. The system can follow either the EBCDIC or a user-defined collating sequence.
**WRITE Statement:** Use the FROM phrase of this statement to write DBCS or DBCS/SBCS data to a record. This phrase moves the data in the same manner as the REWRITE statement. (See "REWRITE Statement.")

You must include the shift control characters when you write the data into a device file.

**Data Manipulation Statements**

**Arithmetic Statements:** Because COBOL treats DBCS characters in the same manner that it treats alphanumeric characters, do not use DBCS characters in numeric operations, nor manipulate them with arithmetic statements.

**INSPECT Statement:** You can use any DBCS or DBCS/SBCS data item as an operand for the INSPECT statement. The system tallies and replaces on each half of a DBCS character, including the shift control characters in these operations. Therefore, the data may not be matched properly.

You can use any combination of double-byte character and alphanumeric operands and double-byte character literals or data items. If you use the REPLACING phrase, you might cause parts of the inspected item to be replaced by alphanumeric data, or vice versa.

You cannot replace a character string with a string of a different length. Consider this when replacing alphanumeric characters with DBCS characters, or vice versa.

If you want to control the use of the INSPECT statement with items containing DBCS characters, define data items containing shift control characters. Use the shift-out and shift-in characters as BEFORE/AFTER operands in the INSPECT statement.

The following example shows how you can use the INSPECT statement to replace one DBCS character with another.

```
01 SUBJECT-ITEM PICTURE X(50).
01 DBCS-CHARACTERS VALUE "0E1K20F".
   05 SHIFT-OUT PICTURE X.
   05 DBCS-CHARACTER-1 PICTURE XX.
   05 DBCS-CHARACTER-2 PICTURE XX.
   05 SHIFT-IN PICTURE X.
```

The INSPECT statement would be coded as follows:

```
INSPECT SUBJECT-ITEM
   REPLACING ALL DBCS-CHARACTER-1
     BY DBCS-CHARACTER-2
   AFTER INITIAL SHIFT-OUT.
```

**Note:** Using the AFTER INITIAL SHIFT-OUT phrase helps you to avoid the risk of accidentally replacing two consecutive alphanumeric characters that have the same EBCDIC values as DBCS-CHARACTER-1 (in cases where SUBJECT-ITEM contains DBCS/SBCS data).
You can also use the INSPECT statement to determine if a data item contains DBCS characters, so that appropriate processing can occur. For example:

01 SUBJECT-FIELD PICTURE X(5).
01 TALLY-FIELD PICTURE 9(3) COMP.
01 SHIFTS VALUE "0E0F".
   05 SHIFT-OUT PICTURE X.
   05 SHIFT-IN PICTURE X.

In the Procedure Division you might enter the following:

MOVE ZERO TO TALLY-FIELD.
INSPECT SUBJECT-FIELD TALLYING TALLY-FIELD FOR ALL SHIFT-OUT.
IF TALLY-FIELD IS GREATER THAN ZERO THEN
   PERFORM DBCS-PROCESSING
ELSE
   PERFORM A-N-K-PROCESSING.

**MOVE Statement:** All DBCS characters are moved as alphanumeric character strings. The system does not convert the data or examine it.

You can move DBCS/SBCS literals to group items and alphanumeric items.

If the length of the receiving field is different from that of the sending field, COBOL does one of the following:

- Truncates characters from the sending item if it is longer than the receiving item. This operation can reduce data integrity.
- Pads the sending item with blanks if it is shorter than the receiving item.

To understand more about the effect of editing symbols in the PICTURE clause of the receiving data item, see the COBOL/400 Reference.

**SET Statement (Condition-Name Format):** When you set the condition name to TRUE on this statement, COBOL moves the literal from the VALUE clause to the associated data item. You can move a literal with DBCS characters.

**STRING Statement:** You can use the STRING statement to construct a data item that contains DBCS or DBCS/SBCS subfields. All data in the source data items or literals, including shift control characters, is moved to the receiving data item, one-half of a DBCS character at a time.

**UNSTRING Statement:** The UNSTRING statement treats DBCS data and DBCS/SBCS data the same as alphanumeric data. The UNSTRING operation is performed on one-half of a DBCS character at a time.

Data items can contain both alphanumeric and DBCS characters within the same field.

Use the DELIMITED BY phrase to locate double-byte and alphanumeric subfields within a data field. Identify the data items containing shift control characters, and use those data items as identifiers on the DELIMITED BY phrase. See the following examples for more information on how to do this. Use the POINTER variable to continue scanning through subfields of the sending field.
After the system performs the UNSTRING operation, you can check the delimiters stored by the DELIMITER IN phrases against the shift control character values to see which subfields contain DBCS and which contain alphanumeric characters.

The following example shows how you might set up fields to prepare for the unstring operation on a character string that contain DBCS/SBCS data:

```
/zerodot1 SUBJECT-FIELD PICTURE X(4/zerodot)
/zerodot1 FILLER.
/zerodot5 UNSTRING-TABLE OCCURS 4 TIMES.
1/zerodot RECEIVER PICTURE X(4/zerodot).
1/zerodot DELIMTR PICTURE X.
1/zerodot COUNTS PICTURE 99 COMP.
/zerodot1 SHIFTS VALUE "/zerodot E/zerodot F/zerodot E/zerodot F".
/zerodot5 SHIFT-OUT PICTURE X.
/zerodot5 SHIFT-IN PICTURE X.
```

Code the UNSTRING statement as follows:

```
UNSTRING SUBJECT-FIELD DELIMITED BY SHIFT-OUT
    OR SHIFT-IN
INTO RECEIVER (1) DELIMITER IN DELIMTR (1)
    COUNT IN COUNTS (1)
INTO RECEIVER (2) DELIMITER IN DELIMTR (2)
    COUNT IN COUNTS (2)
INTO RECEIVER (3) DELIMITER IN DELIMTR (3)
    COUNT IN COUNTS (3)
INTO RECEIVER (4) DELIMITER IN DELIMTR (4)
    COUNT IN COUNTS (4)
ON OVERFLOW PERFORM UNSTRING-OVERFLOW-MESSAGE.
```

This UNSTRING statement divides a character string into its alphanumeric and DBCS parts. Assuming that the data in the character string is valid, a delimiter value of shift-out indicates that the corresponding receiving field contains alphanumeric data, while a value of shift-in indicates that corresponding receiving field has DBCS data. You can check the COUNT data items to determine whether each receiving field received any characters. The following figure is an example that shows the results of the UNSTRING operation just described:

```
SUBJECT-FIELD = ABC/zerodot E/zerodot K1K2K3/zerodot F/zerodot D/zerodot E/zerodot K4K5K6/zerodot F
RECEIVER (1) = ABC DELIMTR (1) = 0E COUNTS (1) = 3
RECEIVER (2) = K1K2K3 DELIMTR (2) = 0F COUNTS (2) = 6
RECEIVER (3) = D DELIMTR (3) = 0E COUNTS (3) = 1
RECEIVER (4) = K4K5K6 DELIMTR (4) = 0F COUNTS (4) = 6
```

```
SUBJECT-FIELD = /zerodot E/zerodot K1K2K3/zerodot F/zerodot ABC/zerodot E/zerodot K4/zerodot F
RECEIVER (1) = (blanks) DELIMTR (1) = 0E COUNTS (1) = 0
RECEIVER (2) = K1K2K3 DELIMTR (2) = 0F COUNTS (2) = 6
RECEIVER (3) = ABC DELIMTR (3) = 0E COUNTS (3) = 3
RECEIVER (4) = K4 DELIMTR (4) = 0F COUNTS (4) = 2
```
Procedure Branching Statements
You can use either a DBCS or a DBCS/SBCS literal as the operand for the STOP statement. When you do, the system displays the literal as you entered it at your work station for interactive jobs. For batch jobs, the system displays underscores where the literal would normally appear on the system operator’s message queue. The system does not edit or check the contents of the literal.

Table Handling–SEARCH Statement
You can perform a Format 1 SEARCH statement (sequential search of a table) on a table that contains DBCS or DBCS/SBCS data half a DBCS character at a time.

You can also perform a Format 2 SEARCH statement (SEARCH ALL) against a DBCS or DBCS/SBCS table as well. Order the table according to the chosen collating sequence.

Note: The shift control characters in DBCS and DBCS/SBCS data participate in the comparison.

SORT/MERGE
You cannot perform a DBCS alphabet sort using COBOL. However, you can use DBCS or DBCS/SBCS data items as keys in a SORT or MERGE statement. The sort operation orders data according to the collating sequence specified in the SORT, MERGE, or SPECIAL NAMES paragraph. The system orders any shift control characters contained in DBCS and DBCS/SBCS keys.

Use the RELEASE statement to transfer records containing DBCS characters from an input/output area to the initial phase of a sort operation. The system performs the FROM phrase with the RELEASE statement in the same way it performs the FROM phrase with the WRITE statement. (See “WRITE Statement” on page 345.)

You can also use the RETURN statement to transfer records containing DBCS characters from the final phase of a sort or merge operation to an input/output area. The system performs the INTO phrase with the RETURN statement in the same manner that it performs the INTO phrase with the READ statement. (See “READ Statement” on page 344.)

Compiler-Directing Statements

COPY Statement
You can use the COPY statement to copy source text that contains DBCS characters into a COBOL program. When you do, make sure that you specify the member name, file name, and library name using alphanumeric data, and that you specify these names according to the rules stated in the COBOL/400 Reference.

Use the Format 2 COPY statement to copy fields defined in the data description specifications (DDS). DBCS and DBCS/SBCS data items (the value in column 35 of the DDS form is O) are copied into a COBOL program in the PICTURE X(n) format. The compiler listing does not indicate that these fields contain DBCS characters, unless a field is a key field. In those cases, the system prints an O in the comment table for keys.

DBCS-graphic data items are copied into a COBOL program in the PICTURE X(N) format. The compiler listing indicates that these fields contain graphic data. See
“DBCS-Graphic Fields” on page 133 for a description of the DBCS-graphic data type.

You can put DBCS characters in text comments that are copied from DDS if the associated DDS field has comments.

If you specify the REPLACING phrase of the COPY statement, consider the following:

- Pseudo-text can contain any combination of DBCS and alphanumeric characters.
- You can use literals with DBCS or DBCS/SBCS content.
- Identifiers can refer to data items that contain DBCS characters.

**TITLE Statement**
You can use DBCS/SBCS literals as the literal in the TITLE statement.

**Communications between Programs**
You can specify entries for DBCS or DBCS/SBCS data items in the Linkage Section of the Data Division.

You can pass DBCS characters from one program to another program by specifying those data items in the USING phrase. You cannot use DBCS characters in the CALL statement for the program-name of the called program.

You cannot use DBCS characters in the CANCEL statement because they specify program-names.

**FIPS Flagger**
Enhancements to the COBOL language that let you use DBCS characters are flagged (identified) by the FIPS (Federal Information Processing Standard) flagger provided by the compiler as IBM extensions.

**COBOL Program Listings**
DBCS characters can appear in listings that originate from DBCS-capable source files, and that are produced on DBCS-capable systems.

DBCS characters that appear in a program listing originate from the source file, from source text generated by the COPY statement, or from COBOL compiler messages.

A listing containing DBCS characters should be output to a printer file that is capable of processing DBCS data. Listings containing DBCS characters are handled correctly if one of the following conditions is true:

- The printer file specified by the PRTFILE parameter of the CRTCBLPGM command is defined with the required attribute, using the CRTPRTF or CHGPRTF command.
- The source file is defined as capable of containing DBCS data using the IGCDTA parameter of the CRTSRCPF command. In this case, the program overrides the existing value of the attribute for the output printer file.
The user has specified the required attribute for the output printer, using the IGCDTA parameter of the OVRPRTF command, before compiling the program.

**Note:** The IGCDTA parameter is only available on DBCS systems, and it cannot be defined or displayed on non-DBCS systems. You can, however, create objects with DBCS attributes on a non-DBCS system by copying them from a DBCS system. You should check for possible incompatibilities if you do this.

The compiler may use characters from your source program as substitution parameters in compiler and syntax checker messages. The system does not check or edit the substitution parameters. If you do not specify DBCS characters properly, the system may print or display parts of messages incorrectly.

"End of IBM Extension"
This appendix contains sample programs that illustrate the fundamental programming techniques associated with each type of AS/400 file organization. These examples are intended to be used for planning purposes only, and to illustrate the input/output statements necessary for certain access methods. Other COBOL features (the use of the PERFORM statement, for example) are used only incidentally. The programs illustrated are:

- Sequential File Creation
- Sequential File Updating and Extension
- Indexed File Creation
- Indexed File Updating
- Relative File Creation
- Relative File Updating
- Relative File Retrieval.

Sequential File Creation

This program creates a sequential file of employee salary records. The input records are arranged in ascending order of employee number. The output file has the identical order. (An **output file** is a file that is opened in either the output mode or the extend mode.)
Figure 112 (Part 1 of 2). Example of a Sequential File of Employee Salary Records

IDENTIFICATION DIVISION.

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-PROGRAM. IBM-AS/400.

OBJECT-PROGRAM. IBM-AS/400.

05 000000 ENVIRONMENT DIVISION.

05 000010 IDENTIFICATION DIVISION.

05 000020 PROGRAM-ID. CRTSEQ.

05 000030 DATA DIVISION.

05 000040 FILE-CONTROL.

05 000050 WORKING-STORAGE SECTION.

05 000060 DATA DIVISION.

05 000070 FILE STATUS IS OUTPUT-FILE-STATUS.

05 000080 FILE STATUS IS INPUT-FILE-STATUS.

05 000090 FILE SECTION.

05 000100 SELECT INPUT-FILE ASSIGN TO DISK-FILEA.

05 000110 SELECT OUTPUT-FILE ASSIGN TO DISK-FILEB.

05 000120 OPEN INPUT INPUT-FILE.

PROCEDURE DIVISION.

FD INPUT-FILE LABEL RECORDS STANDARD.

FD OUTPUT-FILE LABEL RECORDS STANDARD.

INPUT-OUTPUT SECTION.

SPECIAL-NAMES. CONSOLE IS TYPEWRITER.

ERROR HANDLING IS DONE AFTER EACH I/O STATEMENT.

DUMMY DECLARATIVES TO ENSURE CONTROL IS RETURNED TO THIS PROGRAM WHEN AN ERROR OCCURS DURING FILE PROCESSING.

ERROR HANDLING IS DONE AFTER EACH I/O STATEMENT.

END DECLARATIVES.

MAIN-PROGRAM SECTION.

END PROGRAM.
Sequential File Updating and Extension

This program updates and extends the file created by the CRTSEQ program. The INPUT-FILE and the MASTER-FILE are each read. When a match is found between INPUT-EMPLOYEE-NUMBER and MST-EMPLOYEE-NUMBER, the input record replaces the original record. After the MASTER-FILE is processed, new employee records are added to the end of the file.
Figure 113 (Part 1 of 2). Example of a Sequential File Update Program
The example in Figure 113 on page 354 includes:

A. A FILE STATUS clause so that the program records the status of the most recent I/O request involving INPUT-FILE.

B. A FILE STATUS clause so that the program records the status of the most recent I/O request involving MASTER-FILE.

C. A USE procedure that is run when an I/O error occurs during the processing of INPUT-FILE.

D. A USE procedure that is run when an I/O error occurs during the processing of MASTER-FILE.

File status values and USE procedures play important roles in error handling. For more information, see Chapter 6, “COBOL/400 Exception and Error Handling.”
Indexed File Creation

An indexed file is a file that records the key and the position of each record in a separate part of the file called an index.

This program creates an indexed file of summary records for bank depositors. The key within each indexed file record is INDEX-KEY (the depositor's account number); the input records are ordered in ascending sequence upon this key. Records are read from the input file and transferred to the indexed file record area. The indexed file record is then written.

Figure 114 (Part 1 of 2). Example of an Indexed File Program
Indexed File Updating

This program updates the indexed file created in the CRTIND program, using dynamic access.

The input records contain the key for the record, the depositor name, and the amount of the transaction.

When the input record is read, the program tests for:

- If this is a transaction record (in which case, all fields of the record are filled)
- If this is a record requesting sequential retrieval of a specific generic class (in which case, only the INPUT-GEN-FLD field of the input record contains data).

Random access is used for the updating and printing of the transaction records. Sequential access is used for the retrieval and printing of all records within one generic class.
Figure 115 (Part 1 of 4). Example of an Indexed File Update Program

```cobol
IDENTIFICATION DIVISION.
PROGRAM-ID. UPDTIND.
DATA DIVISION.
FILE STATUS IS PRINT-FILE-STATUS.
SELECT PRINT-FILE ASSIGN TO PRINTER-QSYSPRT.
FILE STATUS IS INPUT-FILE-STATUS.
SELECT INPUT-FILE ASSIGN TO DISK-FILEH.
FILE STATUS IS MASTER-FILE-STATUS.
SELECT MASTER-FILE ASSIGN TO DISK-INDXFILE.
ORGANIZATION IS INDEXED.
ACCESS IS DYNAMIC.
FILE-CONTROL.
INPUT-OUTPUT SECTION.
OBJECT-COMPUTER. IBM-AS400.
SOURCE-COMPUTER. IBM-AS400.
CONFIGURATION SECTION.
IDENTIFICATION DIVISION.
```

```cobol
WORKING-STORAGE SECTION.
LINAGE 12 LINES FOOTING AT 9.
FD PRINT-FILE LABEL RECORDS OMITTED.
FD INPUT-FILE LABEL RECORDS STANDARD.
FD MASTER-FILE LABEL RECORDS STANDARD.
```

```cobol
INPUTEND PICTURE X VALUE SPACE.
FILE-CONTROL.
```

```cobol
000100  SELECT MASTER-FILE ASSIGN TO DISK-INDXFILE
000110  ORGANIZATION IS INDEXED
000120  ACCESS IS DYNAMIC
000130  RECORD KEY IS MASTER-KEY
000140  FILE STATUS IS MASTER-FILE-STATUS.
000150  SELECT INPUT-FILE ASSIGN TO DISK-FILEH
000160  FILE STATUS IS INPUT-FILE-STATUS.
000170  SELECT PRINT-FILE ASSIGN TO PRINTER-QSYSPRT
000180  FILE STATUS IS PRINT-FILE-STATUS.
```

```cobol
READ LINES FROM MASTER-FILE INTO MASTER-RECORD.
```

```cobol
INPUT-RECORD.
```

```cobol
INPUT-AMT PICTURE $9(5)V99.
INPUT-NAME PICTURE X(2/zerodot).
INPUT-DET-FLD PICTURE X(5).
INPUT-KEY.
INPUT-REC.
```

```cobol
MASTER-BAL PICTURE $9(5)V99.
MASTER-NAME PICTURE X(2/zerodot).
MASTER-FLD1 PICTURE X(1/zerodot).
MASTER-DET-FLD PICTURE X(5).
MASTER-GEN-FLD PICTURE X(5).
MASTER-KEY.
MASTER-RECORD.
```

```cobol
FILE-CONTROL.
```

```cobol
FD PRINT-FILE LABEL RECORDS OMITTED.
FD MASTER-FILE LABEL RECORDS STANDARD.
```

```cobol
000200  FILE SECTION.
```

```cobol
000210  FD MASTER-FILE LABEL RECORDS STANDARD.
000220  01 MASTER-RECORD.
```

```cobol
000230  05 MASTER-KEY.
```

```cobol
000240  10 MASTER-GEN-FLD PICTURE X(5).
000250  10 MASTER-DET-FLD PICTURE X(5).
```

```cobol
000260  05 MASTER-FLD1 PICTURE X(10).
```

```cobol
000270  05 MASTER-NAME PICTURE X(20).
```

```cobol
000280  05 MASTER-BAL PICTURE $9(5)V99.
```

```cobol
000290  FD INPUT-FILE LABEL RECORDS STANDARD.
000300  01 INPUT-REC.
```

```cobol
000310  05 INPUT-KEY.
```

```cobol
000320  10 INPUT-GEN-FLD PICTURE X(5).
000330  10 INPUT-DET-FLD PICTURE X(5).
```

```cobol
000340  05 INPUT-NAME PICTURE X(20).
```

```cobol
000350  05 INPUT-AMT PICTURE $9(5)V99.
```

```cobol
000360  FD PRINT-FILE LABEL RECORDS OMITTED.
000370  LINA GE 12 LINES FOOTING AT 9.
000380  01 PRINT-RECORD-1.
```

```cobol
000390  05 PRINT-KEY PICTURE X(10).
```

```cobol
000400  05 FILLER PICTURE X(5).
```

```cobol
000410  05 PRINT-NAME PICTURE X(20).
```

```cobol
000420  05 FILLER PICTURE X(5).
```

```cobol
000430  05 PRINT-BAL PICTURE $$9.99-..
```

```cobol
000440  05 FILLER PICTURE X(7).
```

```cobol
000450  05 PRINT-AMT PICTURE $$9.99-..
```

```cobol
000460  05 FILLER PICTURE X(5).
```

```cobol
000470  05 PRINT-NEW-BAL PICTURE $$9.99-..
```

```cobol
000480  01 PRINT-RECORD-2 PICTURE X(99).
```

```cobol
000490  WORKING-STORAGE SECTION.
```

```cobol
000500  77 MASTER-FILE-STATUS PICTURE XX.
000510  77 INPUT-FILE-STATUS PICTURE XX.
000520  77 PRINT-FILE-STATUS PICTURE XX.
000530  77 LINES-TO-FOOT PICTURE 99.
```

```cobol
000540  01 PAGE-HEAD.
```

```cobol
000550  05 FILLER PICTURE X(38) VALUE SPACES.
000560  05 FILLER PICTURE X(13) VALUE "UPDATE REPORT".
```

```cobol
000570  05 FILLER PICTURE X(38) VALUE SPACES.
```

```cobol
000580  01 COLUMN-HEAD.
```

```cobol
000590  05 FILLER PICTURE X(6) VALUE "KEY ID".
```

```cobol
000600  05 FILLER PICTURE X(9) VALUE SPACES.
```

```cobol
000610  05 FILLER PICTURE X(4) VALUE "NAME".
```

```cobol
000620  05 FILLER PICTURE X(21) VALUE SPACES.
```

```cobol
000630  05 FILLER PICTURE X(11) VALUE "CUR BALANCE".
```

```cobol
000640  05 FILLER PICTURE X(6) VALUE SPACES.
```

```cobol
000650  05 FILLER PICTURE X(13) VALUE "UPDATE AMOUNT".
```

```cobol
000660  05 FILLER PICTURE X(4) VALUE SPACES.
```

```cobol
000670  05 FILLER PICTURE X(11) VALUE "NEW BALANCE".
```

```cobol
000680  05 FILLER PICTURE X(4) VALUE SPACES.
```

```cobol
000690  01 PAGE-FOOT.
```

```cobol
000700  05 FILLER PICTURE X(81) VALUE SPACES.
```

```cobol
000710  05 FILLER PICTURE A(6) VALUE "PAGE ".
```

```cobol
000720  05 PG-NUMBER PICTURE 99 VALUE 00.
```

```cobol
000730
```

```cobol
000740  01 INPUTEND PICTURE X VALUE SPACE.
```

```cobol
000750  88 THE-END-OF-INPUT VALUE "E".
```

```cobol```

Figure 115 (Part 1 of 4). Example of an Indexed File Update Program
Figure 115 (Part 2 of 4). Example of an Indexed File Update Program
Figure 115 (Part 3 of 4). Example of an Indexed File Update Program
Relative File Creation

This program creates a relative file of summary sales records using sequential access. Each record contains a five-year summary of unit and dollar sales for one week of the year; there are 52 records within the file, each representing one week.

Each input record represents the summary sales for one week of one year. The records for the first week of the last five years (in ascending order) are the first five input records. The records for the second week of the last five years are the next five input records, and so on. Thus, five input records fill one output record.

The RELATIVE KEY for the RELATIVE-FILE is not specified because it is not required for sequential access unless the START statement is used. (For updating, however, the key is INPUT-WEEK.)
Figure 116 (Part 1 of 2). Example of a Relative File Program
Relative File Updating

This program uses sequential access to update the file of summary sales records created in the CRTREL program. The updating program adds a record for the new year and deletes the oldest year's records from RELATIVE-FILE.

The input record represents the summary sales record for one week of the preceding year. The RELATIVE KEY for the RELATIVE-FILE is in the input record as INPUT-WEEK. The RELATIVE KEY is used to check that the record was correctly written.
Figure 117 (Part 1 of 2). Example of a Relative File Update Program
Relative File Retrieval

This program retrieves the summary file created by the CRTREL program, using dynamic access.

The records of the INPUT-FILE contain one required field (INPUT-WEEK), which is the RELATIVE KEY for RELATIVE-FILE, and one optional field (END-WEEK). An input record containing data in INPUT-WEEK and spaces in END-WEEK requests a printout for that one specific RELATIVE-RECORD; the record is retrieved through random access. (Random processing is a method of processing in which records can be read from, written to, or removed from a file in an order requested by the program that is using them.) An input record containing data in both INPUT-WEEK and END-WEEK requests a printout of all the RELATIVE-FILE records within the RELATIVE KEY range of INPUT-WEEK through END-WEEK inclusive. These records are retrieved through sequential access.
Figure 118 (Part 1 of 2). Example of a Relative File Retrieval Program
5763CB1 V3R0MS AS/400 COBOL Source

STMT SEQNBR -A 1 B..+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME CHG DATE

000750 MAIN-PROCEDURE SECTION.
000760 MAIN-PROCESSING.

62 000770 MOVE "OPEN" TO OP-NAME.
63 000780 OPEN INPUT INPUT-FILE RELATIVE-FILE
000790 OUTPUT PRINT-FILE.
64 000800 MOVE SPACES TO PRINT-RECORD.
65 000810 PERFORM READ-INPUT-FILE.
66 000820 PERFORM CONTROL-PROCESS THRU READ-INPUT-FILE
000830 UNTIL THE-END-OF-INPUT.
67 000840 MOVE "CLOSE" TO OP-NAME.
68 000850 CLOSE RELATIVE-FILE
000860 INPUT-FILE
000870 PRINT-FILE.
69 000880 STOP RUN.
000890 CONTROL-PROCESS.
70 000900 IF (END-WEEK = SPACES OR END-WEEK = 00)
71 000910 PERFORM RANDOM-PROCESS
000920 ELSE
72 000930 PERFORM SEQUENTIAL-PROCESS.
000940 READ-INPUT-FILE.
73 000950 MOVE "READ" TO OP-NAME.
74 000960 READ INPUT-FILE
75 000970 AT END SET THE-END-OF-INPUT TO TRUE.
000980 RANDOM-PROCESS.
76 000990 MOVE "READ" TO OP-NAME.
77 001000 READ RELATIVE-FILE
78 001010 INVALID KEY MOVE HIGH-WEEK TO RELATIVE-WEEK(1).
79 001020 IF RELATIVE-WEEK(1) NOT EQUAL HIGH-WEEK
80 001030 PERFORM PRINT-SUMMARY VARYING REL-INDEX FROM 1 BY 1
001040 UNTIL REL-INDEX > 5.
001050 SEQUENTIAL-PROCESS.
81 001060 MOVE "READ" TO OP-NAME.
82 001070 READ RELATIVE-FILE
83 001080 INVALID KEY MOVE HIGH-WEEK TO RELATIVE-WEEK(1).
84 001090 PERFORM READ-REL-SEQ
001100 UNTIL RELATIVE-WEEK(1) GREATER THAN END-WEEK.
001110 001120 READ-REL-SEQ.
85 001130 PERFORM PRINT-SUMMARY VARYING REL-INDEX FROM 1 BY 1
001140 UNTIL REL-INDEX > 5.
86 001150 MOVE "READ NEXT" TO OP-NAME.
87 001160 READ RELATIVE-INDEX NEXT RECORD
88 001170 AT END MOVE HIGH-WEEK TO RELATIVE-WEEK(1).
001180 PRINT-SUMMARY.
89 001190 MOVE RELATIVE-YEAR (REL-INDEX) TO PRINT-YEAR.
90 001200 MOVE RELATIVE-WEEK (REL-INDEX) TO PRINT-WEEK.
91 001210 MOVE RELATIVE-UNIT-SALES (REL-INDEX) TO PRINT-UNIT-SALES.
92 001220 MOVE RELATIVE-DOLLAR-SALES(REL-INDEX) TO PRINT-DOLLAR-SALES.
93 001230 MOVE "WRITE" TO OP-NAME.
94 001240 WRITE PRINT-RECORD AFTER ADVANCING 2 LINES.

Figure 118 (Part 2 of 2). Example of a Relative File Retrieval Program
Sorting and Merging Files

Figure 119 illustrates the creation of sorted files of current sales and year-to-date sales.

First, the SORT statement for current sales is executed. The input procedure for this sorting operation is SCREEN-DEPT. The records are sorted in ascending order of department, and within each department, in descending order of net sales. The output for this sort is then printed.

After the sorting operation is completed, the current sales records are merged with the year-to-date sales records. The records in this file are merged in ascending order of department number and, within each department, in ascending order of employee numbers, and, for each employee, in ascending order of months to create an updated year-to-date master file.

When the merging process finishes, the updated year-to-date master file is printed.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SORTMERGE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
REQUESTOR IS CONSOLE.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT WORK-FILE ASSIGN TO DISK-WRK.
SELECT CURRENT-SALES-FILE-IN ASSIGN TO DISK-CURRIN.
SELECT CURRENT-SALES-FILE-OUT ASSIGN TO DISK-CURROUT.
SELECT YTD-SALES-FILE-IN ASSIGN TO DISK-YTDIN.
SELECT YTD-SALES-FILE-OUT ASSIGN TO DISK-YTDOUT.
SELECT PRINTER-OUT ASSIGN TO PRINTER-QPRINT.
DATA DIVISION.
FILE SECTION.
SD WORK-FILE.
DATA RECORD IS SALES-RECORD.
FD CURRENT-SALES-FILE-IN.
DATA RECORD CURRENT-SALES-IN.
FD CURRENT-SALES-FILE-OUT.
DATA RECORD CURRENT-SALES-OUT.
```

Figure 119 (Part 1 of 3). Example of Use of SORT/MERGE
Figure 119 (Part 2 of 3). Example of Use of SORT/MERGE
5763CBI V3R0M5 910524     AS/400 COBOL Source
STMT SEQNR -A 1 B...-...4-..5...6...7...8...IDENTFCN S COPYNAME CHG DATE
96 001250 MOVE "UNSORTED CURRENT SALES ",
001260 TO RECORD-LABEL OF PRINT-LINE.
97 001270 MOVE CURRENT-SALES-IN TO DISK-RECORD-DISPLAY.
98 001280 WRITE PRINT-LINE.
99 001290 IF ON-SITE-EMPLOYEE
100 001300 MOVE CURRENT-SALES-IN TO SALES-RECORD
101 001310 RELEASE SALES-RECORD.
001320 080-END-READ-SELECT-DEPT.
001330 EXIT.
102 001340 090-END-S-0-1.
001350 100-END-SCREEN-DEPT SECTION.
001360 100-PRINT-SALES-FILE-OUT.
103 001370 READ CURRENT-SALES-FILE-OUT
104 001380 AT END MOVE "T" TO SALES-FILE-OUT-EOF-STATUS
105 001390 GO TO 110-END-PRINT-SALES-FILE-OUT.
106 001400 MOVE "SORTED CURRENT SALES ",
001410 TO RECORD-LABEL OF PRINT-LINE.
107 001420 MOVE CURRENT-SALES-OUT TO DISK-RECORD-DISPLAY.
108 001430 WRITE PRINT-LINE.
001440 110-END-PRINT-SALES-FILE-OUT.
001450 EXIT.
109 001460 120-READ-PRINT-YTD-SALES-OUT.
110 001470 READ YTD-SALES-FILE-OUT
111 001480 AT END MOVE "T" TO YTD-SALES-OUT-EOF-STATUS
112 001490 GO TO 130-END-READ-PRT-YTD-SALES-OUT.
113 001500 MOVE "MERGED YTD SALES ",
001510 TO RECORD-LABEL OF PRINT-LINE.
114 001520 MOVE YTD-SALES-OUT TO DISK-RECORD-DISPLAY.
115 001530 WRITE PRINT-LINE.
001540 130-END-READ-PRT-YTD-SALES-OUT.
001550 EXIT.

Figure 119 (Part 3 of 3). Example of Use of SORT/MERGE
Appendix H. Example of a COBOL Formatted Dump

Figure 120 on page 372 shows an example of a COBOL formatted dump. To ensure that a dump is available if something goes wrong when you try to run your program, change the INQMSGRPY parameter of the job (for instance, by using the CHGJOB command) to "RQD. When prompted, you can then specify that a dump be generated.

The following list describes the labeled areas of the figure:

- **A** The exception for which the dump was requested and the location in the program where the exception occurred.
- **B** The COBOL statement number of the last I-O operation that was run before the exception occurred. This information is produced only if at least one I-O operation has been processed.
- **C** The current information for each file. This information is produced only if the program has files.
- **D** Beginning of compiler-generated fields (included in the dump if you respond with an F option).
- **E** I-O flags for the current file:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>File is open</td>
</tr>
<tr>
<td>2</td>
<td>File is locked</td>
</tr>
<tr>
<td>3</td>
<td>End of file</td>
</tr>
<tr>
<td>4</td>
<td>(Reserved)</td>
</tr>
<tr>
<td>5</td>
<td>Optional file</td>
</tr>
<tr>
<td>6</td>
<td>Check indexed file for duplicates at open</td>
</tr>
<tr>
<td>7</td>
<td>End of page</td>
</tr>
<tr>
<td>8</td>
<td>(Reserved)</td>
</tr>
</tbody>
</table>

- **F** Previous status code.
- **G** Beginning of Module Global Table (MGT).³
- **H** Last exception code.
- **I** Invocation number of current program.
- **J** Qualified program name and library.
- **K** Beginning of the Program Global Table (PGT).⁴
- **L** Invitation number of the main COBOL program.
- **M** Job date (YYMMDD).
- **N** Beginning of user fields.
- **O** Invalid zoned field printed in hexadecimal.

³ The Module Global Table (MGT) defines a common area for the module. The table is used to pass information to run-time subroutines.

⁴ The Program Global Table (PGT) is a communication area for the entire COBOL run unit. There is only one PGT for the run unit.
Figure 120 (Part 1 of 10). Example of a COBOL Formatted Dump

```
IDENTIFICATION DIVISION.

PROGRAM-ID. XMPLDUMP.

DATA DIVISION.

FILE SECTION.
SELECT FILE-1 ASSIGN TO DISK-SALES.

OBJECT-COMPUTER. IBM-AS/400.

INSTALLATION. COBOL DEVELOPMENT CENTRE.

CONFIGURATION SECTION.

SOURCE-COMPUTER. IBM-AS/400.

IDENTIFICATION DIVISION.

PROGRAM-ID. XMPLDUMP.

001000 IDENTIFICATION DIVISION.

002000 PROGRAM-ID. XMPLDUMP.

003000 AUTHOR.

004000 INSTALLATION. COBOL DEVELOPMENT CENTRE.


006000 DATE-UPDATED. 05/24/94 12:21:54.

007000 ENVIRONMENT DIVISION.

008000 CONFIGURATION SECTION.

009000 SOURCE-COMPUTER. IBM-AS/400.

010000 OBJECT-COMPUTER. IBM-AS/400.

011000 INPUT-OUTPUT SECTION.

012000 FILE-CONTROL.

013000 SELECT FILE-1 ASSIGN TO DISK-SALES.

014000 DATA DIVISION.

015000 FILE SECTION.

016000 FD FILE-1.

017000 LABEL RECORDS ARE STANDARD.

018000 01 RECORD-1.

019000 05 R-TYPE PIC X(1).

020000 05 R-AREA-CODE PIC 9(2).

021000 88 R-NORTH-EAST VALUES IS THROUGH 30.

022000 05 R-SALES-CAT-1 PIC S9(5)V9(2) COMP-3.

023000 05 R-SALES-CAT-2 PIC S9(5)V9(2) COMP-3.

024000 05 FILLER PIC X(1).

025000

026000 WORKING-STORAGE SECTION.

027000 01 W-SALES-VALUES.

028000 05 W-CAT-1 PIC S9(8)V9(2).

029000 05 W-CAT-2 PIC S9(8)V9(2).

030000 05 W-TOTAL PIC S9(8)V9(2).

031000

032000 01 W-EDIT-VALUES.

033000 05 FILLER PIC X(8) VALUE "TOTALS: ".

034000 05 W-EDIT-1 PIC Z(7).9(2).

035000 05 FILLER PIC X(3) VALUE SPACES.

036000 05 W-EDIT-2 PIC Z(7).9(2).

037000 05 FILLER PIC X(3) VALUE SPACES.

038000 05 W-EDIT-TOTAL PIC Z(7).9(2).

039000

040000 01 END-FLAG PIC X(1) VALUE SPACE.

041000 88 END-OF-INPUT PIC S9(8)9(2).

042000

043000 PROCEDURE DIVISION.

044000 ************************************************************

045000 OPEN THE INPUT FILE, CLEAR TOTALS, CALL MAIN PROCESS THEN *

046000 DISPLAY THE RESULTS AND END THE RUN. *

047000 ************************************************************

048000 P-START.

049000 OPEN INPUT FILE-1.

050000 MOVE ZERO TO W-SALES-VALUES.

051000 PERFORM P-MAIN UNTIL END-OF-INPUT.

052000

053000 MOVE W-CAT-1 TO W-EDIT-1.

054000 MOVE W-CAT-2 TO W-EDIT-2.

055000 MOVE W-TOTAL TO W-EDIT-TOTAL.

056000 DISPLAY W-EDIT-VALUES.

057000 STOP RUN.

058000

059000 ************************************************************

060000 READ THE INPUT FILE PROCESSING ONLY THOSE RECORDS FOR THE *

061000 NORTH EAST AREA. WHEN END-OF-INPUT REACHED, SET THE FLAG +

062000 ************************************************************

063000 P-MAIN.

064000 READ FILE-1 AT END SET END-OF-INPUT TO TRUE.

065000 IF R-NORTH-EAST AND NOT END-OF-INPUT *

066000 ADD R-SALES-CAT-1 TO W-CAT-1, W-TOTAL *

067000 ADD R-SALES-CAT-2 TO W-CAT-2, W-TOTAL.

** ** END OF SOURCE ** **
```

MCH1202 exception in program XMPLDUMP in OTEMIP at M1 instruction number 005C COBOL statement number 51. A
Last I-O operation was at statement 48. B
LBE79003-Information pertaining to file FILE-1. C
LBE79005-File is open.
LBE79006-Last I-O operation completed for file was READ.
LBE7907-Last file status for file was 04.
LBE7910-Last extended file status for file was.

Figure 120 (Part 1 of 10). Example of a COBOL Formatted Dump
Figure 120 (Part 2 of 10). Example of a COBOL Formatted Dump
Figure 120 (Part 3 of 10). Example of a COBOL Formatted Dump
Figure 120 (Part 4 of 10). Example of a COBOL Formatted Dump
Figure 120 (Part 5 of 10). Example of a COBOL Formatted Dump
Figure 120 (Part 6 of 10). Example of a COBOL Formatted Dump
Figure 120 (Part 7 of 10). Example of a COBOL Formatted Dump
Figure 120 (Part 8 of 10). Example of a COBOL Formatted Dump
Figure 120 (Part 9 of 10). Example of a COBOL Formatted Dump
Appendix H. Example of a COBOL Formatted Dump

Figure 120 (Part 10 of 10). Example of a COBOL Formatted Dump
Bibliography

For additional information about topics related to COBOL/400 programming on the AS/400 system, refer to the following IBM AS/400 publications:

- **Communications: Management Guide**, SC41-0024
  Short title: Communications Management Guide

- **Device Configuration Guide**, SC41-8106
  Short title: Device Configuration Guide

- **Software Installation**, SC41-3120
  Short title: Software Installation

- **System Programmer’s Interface Reference**, SC41-8223
  Short title: System Programmer’s Interface Reference

- **Database Guide**, SC41-9659
  Short title: DDS Reference

- **Data Description Specifications Coding Form**, SX41-9891
  Short title: DDS Coding Form

- **Communications: Intersystem Communications Function Programmer’s Guide**, SC41-9590
  Short title: ICF Programmer’s Guide

- **System Operation**, SC41-3203
  Short title: System Operation

- **Basic Security Guide**, SC41-0047 and Security Reference, SC41-8083
  Short titles: Basic Security Guide and Security Reference

- **Distributed Data Management Guide**, SC41-9600
  Short title: DDM Guide

- **Database Guide**, SC41-9659
  Short title: Database Guide

  Short title: IDDU User’s Guide

- **System Programmer’s Interface Reference**, SC41-8223
  Short title: System Programmer’s Interface Reference

- **CICS/400 Application Programming Guide**, SC33-0822
  Short title: CICS/400 Application Programming Guide

- **Communications: Remote Work Station Guide**, SC41-0002
  Short title: Remote Work Station Guide

- **Advanced Backup and Recovery Guide**, SC41-8079
  Short title: Advanced Backup and Recovery Guide

- **Programming: Control Language Programmer’s Guide**, SC41-8077
  Short title: CL Programmer’s Guide

- **New User’s Guide**, SC41-8211
  Short title: New User’s Guide

- **Programming: Control Language Reference**, SC41-0030
  Short title: CL Reference

- **Publications Guide**, GC41-9678
  Short title: Publications Guide

- **Programming: Work Management Guide**, SC41-8078
  Short title: Work Management Guide

- **Systems Application Architecture* Structured Query Language/400 Reference**, SC41-9608
  Short title: SQL/400* Reference

- **Data Management Guide**, SC41-9658
  Short title: Data Management Guide

- **COBOL/400 Reference**, SC09-1813
  Short title: COBOL/400 Reference


For information about Common Programming Interface (CPI) COBOL, refer to the following publication:

### Glossary of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appl Dev Tools</td>
<td>Application Development Tools</td>
<td>Consisting of programs for the AS/400 system, such as the Screen Design Aid (SDA) and the Source Entry Utility (SEU).</td>
</tr>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
<td>An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
<td>American National Standard Code for Information Interchange</td>
<td>The code developed by American National Standards Institute for information exchange among data processing systems, data communications systems, and associated equipment. The ASCII character set consists of 8-bit characters, consisting of 7-bit control characters and symbolic characters, plus one parity-check bit.</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Meaning</td>
<td>Explanation</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CICS</td>
<td>Customer Information</td>
<td>An IBM licensed program that enables transactions entered at remote work stations to be processed concurrently by user-written application programs. The licensed program includes functions for building, using, and maintaining databases, and for communicating with CICS on other operating systems.</td>
</tr>
<tr>
<td></td>
<td>Control Service</td>
<td></td>
</tr>
<tr>
<td>CL</td>
<td>Control Language</td>
<td>The set of all commands with which a user requests system functions.</td>
</tr>
<tr>
<td>DBCS</td>
<td>Double-Byte Character</td>
<td>A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Because each character requires 2 bytes, the typing, displaying, and printing of DBCS characters requires hardware and programs that support DBCS. Four double-byte character sets are supported by the system: Japanese, Korean, Simplified Chinese, and Traditional Chinese. Contrast with single-byte character set.</td>
</tr>
<tr>
<td></td>
<td>Set</td>
<td></td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Meaning</td>
<td>Explanation</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>DDM</td>
<td>Distributed Data Management</td>
<td>A function of the operating system that allows an application program or user on one system to use data files stored on remote systems. The systems must be connected by a communications network, and the remote systems must also be using DDM.</td>
</tr>
<tr>
<td>DDS</td>
<td>Data Description Specifications</td>
<td>A description of the user’s database or device files that is entered into the system in a fixed form. The description is then used to create files.</td>
</tr>
<tr>
<td>EBCDIC</td>
<td>Extended Binary-Coded Decimal Interchange Code.</td>
<td>A coded character set consisting of 256 eight-bit characters.</td>
</tr>
<tr>
<td>FIPS</td>
<td>Federal Information Processing Standard</td>
<td>An official standard to improve the utilization and management of computers and data processing in business.</td>
</tr>
<tr>
<td>ICF</td>
<td>Intersystem Communications Function</td>
<td>A function of the operating system that allows a program to communicate interactively with another program or system.</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
<td>Data provided to the computer or data resulting from computer processing.</td>
</tr>
<tr>
<td>LVLCHK</td>
<td>Level Checking</td>
<td>A function that compares the record format-level identifiers of a file to be opened with the file description that is part of a compiled program to determine if the record format for the file changed since the program was compiled.</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Meaning</td>
<td>Explanation</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>ODT</td>
<td>Object Definition Table</td>
<td>A table built at compile time by the system to keep track of objects declared in the program. The program objects in the table include variables, constants, labels, operand lists and exception descriptions. The table resides in the compiled program object.</td>
</tr>
<tr>
<td>SQL/400</td>
<td>Structured Query Language/400</td>
<td>An IBM licensed program supporting the relational database that is used to put information into a database and to get and organize selected information from a database.</td>
</tr>
<tr>
<td>UPSI</td>
<td>User Program Status Indicator switch</td>
<td>An external program switch that performs the functions of a hardware switch. Eight switches are provided: UPSI 0 - 7.</td>
</tr>
</tbody>
</table>

Note: The abbreviations for OS/400 commands do not appear here. Refer to the CL Reference for OS/400 commands and their usage.
Index

Special Characters
/ (slash) 12, 38
   maximum number in a program 89
* (asterisk) 12

Numerics
8-byte binary items, and performance 268

A
abnormal program termination 52
about this manual xi
ACCEPT statement 103, 177, 343
access mode 173, 241, 249
   DYNAMIC 246
   RANDOM 246
access path
   description 128
   example for indexed files 247
   file processing 250, 251
   specifications 105
*ACCUPDALL option 26
*ACCUPDNE option 26
ACQUIRE statement 178
ADDMMSGD (Add Message Description) command 331
ADDRESS OF special register 280, 287
   description 287
   difference from calculated ADDRESS OF 287
addresses
   incrementing using pointers 305
   passing between programs 303
ADM/400
ADVANCING phrase 233
   for FORMATFILES 234
ADVANCING PAGE phrase 336
ALCOBJ (Allocate Object) command 93
ALIAS keyword 111
alias name 113
alias, definition 111
*ALL option 27
Allocate Object (ALCOBJ) command 93
alphabet-name, definition 335
alphabetic character, definition 336
ALTER statement 312
American National Standards Institute (ANSI) xiii,
   1, 323, 331, 385
   ANSI 74 COBOL versus ANSI 85 COBOL 335
   conforming to standards
      with indexed files 241
      with relative files 249
      with sequential files 249
   standard xiii, 1, 331
APIs (Application Programming Interfaces)
   error-handling 53, 70
   using with pointers 291
*APOST option 20
Application Development Manager/400
Application Development Tools, messages 327
Application Programming Interfaces (APIs)
   error-handling 53, 70
   using with pointers 291
arguments, describing in the calling program 280
arithmetic operators 2
arrival sequence 129, 249, 250
arrows, shown in syntax 3
ASSIGN clause 89, 143, 172
   device name 89
assignment name 89, 143, 172
* (asterisk) 12
AT END condition 80, 187, 190
*ATR option 21
ATTRIBUTE DATA 178
attributes
   of data items 46
   of files 45
   of table items 46
ATTRIBUTES field 45
AUT parameter for CRTCBLPGM command 27
authorization-list-name option 27

B
batch compiles 36
batch jobs, representation of DBCS data in 348
binary items, and performance 268
*BLANK option 19
BLANK WHEN ZERO
   defining with LIKE clause 258
*BLK option 23
block, description 102
blocking output records 102
Boolean data types 20, 144, 176
Boolean literal, definition 20
boundary
definition 95
record 23
violation 73, 251
breakpoints
as an OS/400 function 55
considerations for using 63
description 57
displaying table elements 60
displaying variables 60
example 57
traces, differences between 64
use of 57, 62
browsing a compiler listing
See source entry utility (SEU)
BY CONTENT, definition 279
BY REFERENCE, definition 279

C
calculation operations
on fixed-length fields 132
call by identifier 282
CALL statement
BY CONTENT identifier 280
BY CONTENT LENGTH OF identifier 280
BY CONTENT literal 280
BY CONTENT, implicit MOVE 289
by identifier 282
BY REFERENCE ADDRESS OF
record-name 280
BY REFERENCE identifier 279
recursive, description 273
to QCMDEXC 255
using pointers 289
within a segmented program 312
called program
declaration 273
calling programs
BY CONTENT 279
BY REFERENCE 279
definition 273
from a non-COBOL program 268
to begin at another entry point
using pointers 289
within a segmented program 312
calling the COBOL compiler 15
CANCEL statement 282, 307, 336
with non-COBOL programs 279
*CBL statement 38
CCSIDs (Coded Character Set Identiﬁers) 137
CDRA (Character Data Representation Architecture) 137
Change Debug (CHGDBG) command 55
*CHANGE option 27
Change Program Variable (CHGPGMVAR)
command 63
change/date (CHGDATE) ﬁeld 43
changes from ANSI 74 COBOL 335—336
changing the value of variables 63
Character Data Representation Architecture (CDRA) 137
characters, double-byte 337
characters, replaced in ﬁeld name 113
checking DBCS literals 339
checking work station validity 140
CHGDBG (Change Debug) command 55
CHGPGMVAR (Change Program Variable)
command 63
choices, shown in syntax 3
CICS (Customer Information Control System) statements 13
CICSCBL member type 13
CICSSQLCBL member type 13
CL (control language) commands
for running programs 7
for testing programs 55
issuing using QCMDEXC in a program 255
clauses
ACCESS MODE 173
ASSIGN 143, 172
CONTROL-AREA 174
CURRENCY clause 12
DECIMAL-POINT clause 12
FILE STATUS 103, 173
INDICATOR 145
JUSTIFIED 342
LIKE 145
LINAGE 233
OCCURS 145, 341
ORGANIZATION 172
ORGANIZATION IS INDEXED 241
PICTURE 144, 267, 342
RECORD KEY 129
REDEFINES 335, 341
RELATIVE KEY 173
commitment control 82, 94, 98
example 97
locking level 95
common keys 129
Common Programming Interface (CPI)
support 325
communication module 325
communications, interactive
interprogram considerations 273, 349
recovery 83
with other programs 139
with remote systems 139
with work station users 139
COMP-3 items, and performance 268
compile-time errors 56
compiler failure 16
compiler options
See also PROCESS statement
also parameters, CRTCLPGM command
*ACCUPDALL 26
*ACCUPDNE 26
*ALL 27
and syntax checking with SEU 12
*APOST 20
as specified in PROCESS statement 32
*ATR 21
authorization-list-name option 27
batch compiling 36
*BLANK 19
*BLK 23
*CHANGE 27
check for sequence errors 19
check subscript ranges at run time 21
compiler options listing 37, 40
count verb usage 19
create cross-reference listing 19, 21, 47
create Data Division map 20
create object code 19
create source listing 19, 41
*CRTF 22
*CURLIB 18, 25
*CURRENT 26, 31
*DATETIME 24
*DDSFILLER 22
*DEB1 25
*DEB2 25
delimiter for nonnumeric and Boolean
literals 20
*DFRWRT 25
*DUMP 21
compiler options (continued)

-DUPKEYCHK 22
-EXCLUDE 27
-EXTACCDSP 23
file-name option 24
-FLAG 25, 37
-FS21DUPKY 23
-GEN 19
-GRAPHIC option 24
-HIGH 25
include attributes for the IRP 21
-INTERMEDIATE 25
-INZDLT 23
-LIBCRTAUT 27
-LIBL 18, 24
library-name option 18, 25
-LINENUMBER 20
-LIST 21
list compiler options in effect 37, 41
-LSTDBG 21
-MAP 20, 37
maximum-severity-level option 24
message-limit option
-MINIMUM 25
-NOATR 21
-NOBLK 23
-NOCRTF 22
-NODDSFILLER option 24
-NODATETIME 22
-NODEB 25
-NODFRWRT 25
-NODUMP 21
-NODUPKEYCHK 22
-NOEXTACCDSP 23
-NOFIPS 25
-NOFLAG 25
-NOFS21DUPKY 23
-NOGEN 19
-NOINZDLT 23
-NOLIST 21
-NOSTDDBG 21
-NOMAP 20
-NOMAX 24
-NONUMBER 19
-NOOBOLEATE 25
-NOOPTIMIZE 22
-NOOPTIONS 20
-NOPATCH 21
-NOPRINT 21
-NOPRTCORR 20

compiler options (continued)

-NORANGE 22
-NOSECLVL 20
-NOSEG 25
-NOSEQUENCE 19
-NOSEQUE 19
-NOSEQUECN 20
-NOSEQRERR 22
-NOSEQRDINZ 23
-NOSEQRSYNC 22
-NOSEQRDSCHAR 25
-NOSEQRUSEF 22
-NOSEQRVARCHAR 23
-NOSEQRVBSUM 19
-NOSEQRXREF 19, 21
-NUMBER 19
-OBSELETE 25
-OPMIZE 22
optimizing source code 22
-OPTIONS 20, 37
overview 6
-OWNER 26
parameters of the CRTCBLPGM
command 18—31
-PATCH 21
-PGM 18
-PGMID 18
-PRINT 21
PROCESS statement, using to specify 32
program listings, DBCS characters in 349
program-name 18
-PRTCORR 20
-PRV 26, 31
QLBLSRC (default source file) 18
QSBSRT (default printer file) option 24
-QUOTE 20
-RANGE 21
release-level option 26, 31
-SECEVL 20
-SEG1 25
-SEG2 25
-SEQUENCE 19
severity-level option 19, 26
-SOURCE 18, 19, 37
source-file-member-name option 19
source-file-name option 18
specifying
-SRCDBG 20
-SRCMBFTXT 19
-STDERR 23
compiler options (continued)

*STDINZ 23
suppressing second-level messages 20
suppressing source listing 41
*SYNC 22
text-description 19
*UNDSPCHR 25
*UNREF 22
*USE 27
use compiler-generated sequence numbers 20
use user-supplied sequence numbers 19
*USER 26
V2R1M0 value for release-level option 31
V2R1M1 value for release-level option 31
V2R2M0 value for release-level option 31
*VARCHAR 23
*VBSUM 19, 37
*XREF 19, 21, 37

compiler output
See also messages
browsing 39

See also source entry utility
command summary listing 39
compiler output 36, 37
cross-reference listing 47
CRTCBLPGM options 37
Data Division map 44
description 37
tabular 37
FIPS messages listing 46
listing descriptions 37
listing options 40
messages 329
options listing 39, 40
program listings, DBCS characters in 349
suppressing source listing 41

compiling COBOL programs
abnormal compiler termination 16
compiler-detected errors 56
example listing 39
exceeding internal size limits 15
failed attempts 16
for extended ACCEPT/DISPLAY 23
for the previous release 31
invoking the compiler 15
messages 329
multiple programs 36
output 37
redirecting files 90
TGTRLS, using 31

CONCAT keyword 122
Configuration Section, description 10, 340
conforming to ANSI standards 332
constant, NULL figurative 286
contents of DEBUG-ITEM special register 319
contiguous items, definition 242
contiguous key fields, multiple 242
control
returning 274
transferring 273
CONTROL-AREA clause 174—175
control language commands
See CL commands
control of segmentation 310
*CONTROL statement 38
COPY DDS, use with indicators 118
copy, returning from a called program 274
copy, transferring to another program 273
CoOperative Development Environment/400
copies of ANSI standard available xiii
COPY statement
and DBCS characters 348
and externally described data 113
and floating-point 127
changes from ANSI 74 COBOL 335
data field structures 116
DD, DDR, DDS, or DDDR 112
DDS results 110, 118
description 112
data set 46
example of data structures generated by 204
elements of key generation 121
format-1 COPY statement 36
in File Section 114
key fields 242
listing source statements 38
outside File Section 114
PROCESS statement containing COPY state-
ment 37
suppressing source statements 38
use with PROCESS statement 36
use with TRANSACTION files 139
with ALL-FORMATS 114
COPYNAME field 43
corresponding options, PROCESS and
CRTCBLPGM 32
CORRESPONDING phrase 256
counting verbs in a source program 19, 43, 49
CPI (Common Programming Interface)
support 325
Create Authorization List (CRTAUTL) command 27
Create COBOL Program (CRTCBLPGM) command
  AUT parameter 27
  CVTOPT parameter 23, 34
description of 6
DUMP parameter 27
  entering from CL program 28
  entering from command line 28
EXTDSPOPT parameter 35
FLAG parameter 26, 35
FLAGSTD parameter 25, 35, 37
GENLVL parameter 19, 32
GENOPT parameter 21, 34
ITDUMP (n) parameter 27
MSGLMT parameter 24
OPTION parameter 19, 33, 37
  parameters, description of 18—31
PGM parameter 18
  prompt displays, using 16
PRTFILE parameter 24
REPLACE parameter 26
SAAFLAG parameter 25, 35, 37
SRCFILE parameter 18
SRCMBR parameter 18
  syntax of 29
TEXT parameter 19
TGTRLS parameter 26
USRPRF parameter 26
creating files
  indexed files 351, 356
  relative files 351, 361
  sequential files 351
cross-reference listing and breakpoints 57
  CRTCBLPGM options 19, 21
description of listing 48
every example 47
testing, using in 61
CRTAUTL (Create Authorization List) command 27
CRTCBLPGM command
  See Create COBOL Program command
  *CRTF option 22
  *CURLIB option 18, 25
  *CURRENT option 26, 31
Customer Information Control System (CICS) statements 13
CVTOPT parameter 23, 34
D
data area
description 305
  local 305
  PIP 306
data class type (TYPE) field 45
data communications file 139, 172
data description entry for Boolean data 144
data description specifications (DDS) command attention (CA) keys 140
  CONCAT keyword 122
  Create File commands 105
date fields 132
  DD option, description 113
  DDR option, description 113
date fields, description 113
definition 140
description 106
display management 140
data communications file 139, 172
data communications files 139, 172
data communications programs 137
data communications stations 137
data communications systems 137
data communications terminals 137
data communications terminal 137
data communications terminals, description 137
data communications type 137
data communications unit 137
data communications unit, description 137
data communications units 137
data communications units, description 137
data description entry for Boolean data 144
data description specifications (DDS) command attention (CA) keys 140
  CONCAT keyword 122
  Create File commands 105
date fields 132
date fields, description 113
definition 140
description 106
display management 140
data communications file 139, 172
data communications files 139, 172
data communications programs 137
data communications stations 137
data communications systems 137
data communications terminals 137
data communications terminal 137
data communications terminals, description 137
data communications type 137
data communications unit 137
data communications unit, description 137
data communications units 137
data communications units, description 137
data communications unit 137
data communications units 137
data communications units, description 137
data communications type 137
data communications unit 137
data communications units 137
data communications units, description 137
data communications unit 137
data communications units 137
data communications units, description 137
data communications type 137
data communications unit 137
data communications units 137
data communications units, description 137

time fields 132
Index 395
designing your program 9
destination of compiler output 36
device control information 142
device dependence 89
  examples 90
device files
  and I/O 89
  DATABASE file considerations 241
  DISK file considerations 241
  multiple 162
  single 162
device independence 89
*DFRWRT option 25
diagnostic levels 330
diagnostic messages 48
diagrams, syntax 29
direct files
  See relative files
disclaimers
  examples
  patents ix
  sending information to IBM
  US government users
disk files 241
  processing methods 252
displacement (DISP) field 45
display device
  DDS for 140
  record format 140, 141
display device file 140
display format data, definition 140
DISPLAY statement 344
Display Trace Data (DSPTRCDTA) command 65
displaying a compiler listing 39
displays
  CRTCBBLPGM prompt display 17
  data description specifications (DDS) for 140
  display program messages 328
  ENDCBBLDBG prompt display 315
  for sample programs
    order inquiry 216, 217
    payment update 228, 229, 230
    transaction inquiry 205
  SEU display messages 327
  STRCBBLDBG prompt display 314
  subfiles 157
distributed data management (DDM) 387
DIVIDE statement 336
divisions of programs
  Data Division 20, 173, 175, 341
divisions of programs (continued)
  Environment Division 171, 310, 311, 340
  Identification Division 10
  optional 10
  Procedure Division 176, 311, 343—348
  required 10
do while structure, testing for end of chained list 304
double spacing 38
double-byte character set (DBCS)
support 337—350
  ACCEPT statement 343
  and alphanumeric data 346
  checking 339
  comments with DBCS characters 340
  communications between programs 349
  definition 386
  enabling in COBOL programs 337
  graphic 348
  in the Data Division 341
  in the Environment Division 340
  in the Identification Division 340
  in the Procedure Division 343—348
  open 348
  PROCESS statement 337, 345
  representation of DBCS data in batch jobs 348
  searching for in a table 348
  sorting 348
  specifying DBCS literals 337, 338
DROP statement 179
DSPTRCDTA (Display Trace Data) command 65
*DUMP option 21
DUMP parameter for CRTCBBLPGM command 27
dump, formatted 371
*DUPKEYCHK option 22, 270
duplication errors 258
dynamic access mode 158, 173, 249, 253
dynamic file creation 22
dynamic processing, definition 173

E
EBCDIC character, definition 387
editing source programs 9
  See also source entry utility (SEU)
efficiency considerations 268
efficiency, increased 22
eight-byte binary items, and performance 268
EJECT statement 38
elementary pointer data items 287
embedded SQL 12
End COBOL Debug (ENDCBLDBG)
command 315, 316
description for de-editing, run time 267
diagnostic messages listing 48
diagnostic messages listing 48
diagnostic messages listing 48
diagnostic messages listing 48
diagnostic messages listing 48
diagnostic messages listing 48
diagnostic messages listing 48
diagnostic messages listing 48
diagnostic messages listing 48
diagnostic messages listing 48
diagnostic messages listing 48
diagnostic messages listing 48
diagnostic messages listing 48
diagnostic messages listing 48
diagnostic messages listing 48
diagnostic messages listing 48
diagnostic messages listing 48
diagnostic messages listing 48
diagnostic messages listing 48
diagnostic messages listing 48
diagnostic messages listing 48
END-OF-PAGE phrase 336
END-READ phrase 187, 190
END-REWRITE phrase 192
END-WRITE phrase 199
ending a called program 274
entering CRTCBPDM from CL program 28
entering CRTCBPDM from command line 28
entering source programs 9, 11
entering your program
See source entry utility (SEU)
Environment Division
and DBCS characters 340
and transaction files 171
SEGMENT-LIMIT clause 310, 311
error checking for de-editing, run time 267
error handling 69
APIs 53, 70
nonstandard 77
overview 69
standard 76
error recovery, example 82
errors
ADVANCING phrase with FORMATFILE
files 234
duplication 258
errors to avoid 56
errors, in syntax
See syntax errors
examples
access path for indexed file 247
breakpoint 57
COBOL and files 105
commitment control 94, 98
compiler options listing 20, 37
COPY DDS results 110, 119
COPY statement in PROCESS statement 37
cross-reference listing 47
Data Division map 44
DDS
CONCAT keyword 122
for a display device file 140, 141
for a record format 109
for a record format with ALIAS keyword 111
for field reference file 107
for multiple device files 162
examples (continued)
DDS (continued)
for subfiles 159, 161
key generation 121
RENAME keyword 124
SST keyword 126
diagnostic messages listing 48
END-OF-PAGE condition 235
entering CRTCBPDM from command line 28
error recovery 82
externally described printer files 237
file processing
indexed files 356, 357
relative files 361, 363
sequential files 351, 353
FIPS messages listing 46
FORMATFILE file 234
formatted dump 371
generic START 242, 243
indicators 146
LENGTH OF special register with pointers 286
length of variable-length field 132
MOVE with pointers 288
multiple device files 165
pointers
aligning 284
and LENGTH OF special register 286
and REDEFINES clause 285
and results of MOVE 288
initializing with NULL 286
passing items containing 289
processing chained list 302
program structure 9
record format specifications 107, 110
ROLLING phrase 195
run units
multiples, running concurrently 278
multiples, running consecutively 276
single unit 274
with shared program 277
SEU display messages 327
source listing 41
trace 64
using pointers in chained list 302
variable-length graphic data 134
verb usage by count listing 43
work station application programs
order inquiry 206
payment update 217
transaction inquiry 200
exceptions 16, 52, 71, 81
*EXCLUDE option 27
exclusive-allow-read lock state 93
EXIT PROGRAM statement 274, 306
expressions 264, 343
*EXTACCDSPP option 23
EXTDSPOPT parameter of the CRTCLPGM command 35
EXTEND mode, definition 93
extended ACCEPT and DISPLAY statements 23
extensions, IBM
double-byte character set (DBCS) support 337—350
flagging 25, 331
format, indication in syntax 5
GOBACK overview 1
reading 5
transaction files 139—231
extensions, list of 1
external description
adding functions to 130
overriding functions to 130
external file status 70
externally described files 113, 236
adding functions 130
advantages of using for printer files 234
and COPY statement, DD, DDR, DDS, DDDR format 119
considerations for using 105
DDS for 108
description 104
level checking 130
overriding functions 130
printer files, specifying with FORMATFILE 234
EXTERNALLY-DESCRIBED-KEY 242
externally described TRANSACTION files 139—142

Federal Information Processing Standard (FIPS) (continued)
standards to which the compiler adheres xiii
with DBCS characters 349
FIB (file information block) 71
field names
-DDS added to 123, 125
additional notes 127
construction of 116
fields
attributes
BLANK WHEN ZERO 260
defaults 260
SIGN IS TRAILING 260
USAGE IS DISPLAY 260
BLANK WHEN ZERO attribute 260
date 132
fixed length 132
floating-point 127
null-capable 133
time 132
time separator 38
timestamp 132
variable-length 131
character 131
graphic 131, 134
length of, example 132
maximum length 131
restrictions 131
figurative constant QUOTE 20
figurative constant, NULL 286
file and record locking 93, 95
file boundaries 251
file considerations 89, 241, 306
file control entry 89
of Environment Division 171
TRANSACTION file processing entry 171
file descriptions 108, 175
file information block (FIB) 71
file locking 93
file-name option 24
file operations for printer file 233
file organization 250
file processing
See files
file redirection 90, 92
file status
0Q 251
9N 83
9Q 251

F
failed I/O and record locking 94
failure of compiler 16
Federal Information Processing Standard (FIPS)
1986 COBOL standard 331
description 331
flagging deviations from 25, 331, 349
FLAGSTD parameter 25, 46
messages 46, 329, 331
options 25
standard modules 331
file status (continued)
  after I/O 83
  coded examples 353
  from message monitors 73
  how it is set 72
  internal and external 70
  obtaining 103
  statements that affect 306
FILE STATUS clause 103

files
  See also disk files, externally described files,
  program-described files, source files

access paths 250
attributes of 45
closing 336
creation of
  indexed 351, 356
  relative 351, 361
  sequential 351
DATABASE 241
DATABASE versus DISK 241
description 351
DISK 241
examples
  indexed files 356, 357
  relative files 361, 363
  sequential files 351, 353
external description 105
FORMATFILE 234
indexed organization 241
keys 129
logical 246
methods of 251
on AS/400 systems 89, 351
preserving sequence of records 250
PRINTER 233
processing methods 241
redirecting access to 90
relative 249
relative organization 249
retrieval of, relative 351, 365
sample programs 351—367
sequential 249
sequential organization 249
specific 233
techniques for processing 351—367
TRANSACTION 139
files unavailable at OPEN time 22
FILLER
  floating-point fields 127

FIPS flagging
  See Federal Information Processing Standard
FIPS-ID field 46
FIPS violations flagged, total 47
fixed length graphic fields 133
*FLAG option 25, 37
FLAG parameter for CRTCLPGM command 26, 35
FLAGSTD parameter for CRTCLPGM
  command 25, 35, 46
float test, de-editing 267
floating-point fields 127
FOOTING phrase 336
format (record) level structures 115
format-1 COPY statement 36
format-2 COPY statement 16
format names, additional notes 127
FORMAT phrase 181, 186, 189, 191
FORMATFILE files
  description 234
  sample program 234
formats, using SEU
  See source entry utility
formatted dump 55, 67, 371
*FS21DUPKY option 23
function keys
  and CONTROL-AREA clause 175
  specifying with DDS
  See transaction files
functional processing modules 323

G
  *GEN option 19
general-use programming interfaces
    description ix
    error-handling 70
    QCMDEXC 28, 255
generation of I/O formats 118
generation of keys 121
generation of message monitors 73
generic START statement 242
GENLVL parameter for CRTCLPGM
  command 19, 32
GENOPT parameter for CRTCLPGM
  command 21, 34
GIVING phrase 335, 336
GO TO statement 312
GOBACK statement 307
Grant Object Authority (GRTOBJAUT)
command 27
graphic data types 133
restrictions 133
*GRAPHIC option 24
group level names 116
group structures, aligning pointers within 284
GRTOBJAUT (Grant Object Authority)
command 27

H
handling data errors, de-editing 267
*HIGH option 25
highlights 2, 351
hyphen, produced when copying ALIAS names 113

I
I/O formats 118
IBM extensions
  double-byte character set (DBCS) support 337—350
  flagging 25, 331
  format, indication in syntax 5
  GOBACK overview 1
  reading 5
  transaction files 139—231
ICF
  See intersystem communications function
Identification Division
  and DBCS characters 340
description 10
identifier
  call by 282
  defining in Working-Storage section 258
  unreferenced 22
increasing efficiency 22
INDARADDS keyword 118
independence, device 89
independent segments 309
indexed files
  creation 351, 356
description 241
key fields 241
processing methods for types DISK and DATABASE 241
updating 351, 357
indexed files, definition 356
indexed I-O module 324
indicator structures 117
indicators
  and ASSIGN clause 143
  and Boolean data items 144
  and COPY statement 114, 118
  associated with command keys 140
data description entries 144
description 115, 142
element, using in programs 146
in a separate indicator area 143, 145, 270
in the record area 143, 146
INDARA DDS keyword 143
INDICATOR clause 145
INDICATORS phrase 145
performance considerations 270
sample programs 146
special considerations for 144
structures 116
TRANSACTION file processing 142
using 144
industry standards xiii
initialization of storage 279
initializing files with deleted records 251
initializing pointers 286
  with NULL figurative constant 286
input field 140, 189
input records 102
input spool 91, 92
input verbs, processing of
  since Version 1, Release 3 80
Input-Output formats 118
input-output verbs, processing of
  since Version 1, Release 3 80
INSPECT statement 345
inter-program module 324
*INTERMEDIATE option 25
intermediate representation of program (IRP)
cross-reference listing 21
how to list 21
how to list attributes 21
how to use 57
  sample listing 62
internal file status 70
internal name (I-NAME) field 45
internal size limits 15
International Standards Organization (ISO) xiii
interprogram calls using pointers 289
interprogram communication considerations 273
intersystem communications function (ICF)
  ACCESS MODE clause 173
  ASSIGN clause 172
  communications 156
  CONTROL-AREA clause 174
  FILE STATUS clause 173
  multiple and single device files 162
  ORGANIZATION clause 172
  RELATIVE KEY clause 173
  using to specify subfiles 156
  INTO phrase 186, 335
  introduction to COBOL/400 1
  invalid characters 113
    DDR and DDSR options 113
  INVALID KEY phrase 190, 192, 199
    role since Version 1, Release 3 80
  *INZDLT option 23, 251
  I-O feedback 55, 103, 343
  I-O flags 371
  I-O operation 371
  I/O verbs, processing of
    since Version 1, Release 3 80
  IRP (intermediate representation of program)
    See intermediate representation of program (IRP)
  ITDUMP parameter for CRTCBLPGM
    command 27
  items grouped by level 47

J
  job failure, recovery 82
  JUSTIFIED clause 342

K
  key fields
    contiguous, multiple 242
    descending keys 253
    for indexed files 241
    name, -DDS added to 123, 125
    partial keys 242
    program-defined 246
  keyed read 91
  keyed sequence 129, 241, 250, 253
  keys
    common 129
    floating-point 127
    generation of 121
    record 129
  keys (continued)
    validity 242
  keywords
    DDS 111, 122, 124, 126
    in syntax diagrams 2
    INDARA 118

L
  language elements
    See program structure
  last-used state, description 274
  LDA (local data area) 305
  length (LENGTH) field 45
  length of records in source file 10
  LENGTH OF special register 280, 286
  length of statement, maximum 11, 12
  level checking 130
  level of data item (LVL) field 44
  level of language support 323, 324, 331
  *LIBCRTAUT option 27
  *LIBL option 18, 24
  libraries, test 55
  library-name option 18, 25
  LIKE clause
    description 258
    format of 259
    PICTURE portion 260
  limitations 89
    TGTRLS parameter 31
  limits, internal, size 15
  LINAGE clause 233
  *LINENUMBER option 20
  linkage items, setting the address of 287
  Linkage Section
    describing data to be received 281
    parameters for a called program 280
  *LIST option 21
  listings
    browsing
      See source entry utility
    command summary 39
    compiler options in effect 20
    cross-reference 47
    Data Division map 44, 61
    DBCS characters in 349
    default output file 24
    example, source listing 19, 41, 43
    examples of 40
    FIPS messages 46
    messages
      description 48
messages (continued)
  interactive 327
  responding to in an interactive
  environment 329
  run-time 328
  and standard error handling 69
  SAA, flagged 47
  severity levels 19, 24, 330
  statistics 49
  types 327
migrating
  ANSI 74 COBOL programs 335
  to ANSI 85 COBOL 335
  to COBOL/400 language 335
*MINIMUM option 25
mismatched records, reducing occurrence 281
module global table (MGT), definition 371
Monitor Message (MONMSG) command 16
monitoring exceptions 16
monitoring operations
  monitors, message 73
MONMSG (Monitor Message) command 16
MOVE statement 319, 346
  CORRESPONDING phrase 256
  using pointers 287
MSGID and severity level field 49
MSGLMT parameter 24
multiple contiguous key fields 242
multiple device files 162—170, 178, 184, 190
multiple members 92

N
name, assignment 89, 143, 172, 341
names defined when GENOPT(*NOUNREF) speci-
ified 15
NAMES field 48
NEXT MODIFIED phrase 189
NO DATA phrase 186
  role since Version 1, Release 3 80
NO LOCK phrase, and performance 94, 270
NO REWIND phrase 336
*NOATR option 21
*NOBLK option 23
*NOCRTF option 22
*NODATETIME option 24
*NODDSFILLER option 22
*NODEB option 25
null-capable fields 133
NULL figurative constant 286
null values 133, 304
*NUMBER option 19
numeric items
   and performance 268
   moving with de-editing 265
numeric usage clauses 116

O
Object Definition Table (ODT) 22
object names, OS/400 16
object program
   compiler options, specifying 21
   optimizing, specifying at compile-time 22
   output from compiler 15
   specifying authority to 27
   subscript range checking 21
object program, definition 6
*OBsolete option 25
OCCURS clause 341
ODP (open data path) 94
ODT (Object Definition Table) 22
offset, relative to 16-byte boundary 289
open data path (ODP) 94
OPEN operation, increasing speed of 94
OPEN statement 180
OPEN type 93
OPEN-FEEDBACK 343
operation, more efficient 22
operators, arithmetic and logical 2
*OPTIMIZE option 22
optimizing code 22
OPTION parameter for CRTCBPGM
   command 19, 37
optional clauses 3
optional divisions 10
optional items, syntax 3
optional processing modules 324
options
   data formats, *NORANGE option 269
   for the PROCESS statement 36
   listing 40
   of CRTCBPGM command
      parameters 18—31
   segmentation 311
OPTIONS listing 40
*OPTIONS option 20, 33, 37
order of clauses 3
ORGANIZATION clause 172
ORGANIZATION IS INDEXED clause 241
OS/400 operating system
and messages 331
breakpoint commands 57
device control information 142
device independence and device
dependence 89
functions for debugging 55
input/output 142
internal size limits 15
object names 16
security, maintaining while testing 55
testing, functions for 55
output
compiler 37
compiler, displaying 39
output field 140
output file, definition 351
output spool 91
output verbs, processing of
since Version 1, Release 3 80
Override Message File (OVRMSGF)
command 331
Override to Diskette File (OVRDKTF)
command 90
overriding messages 331
overriding program-specified files 92
overview 6
OVRDKTF command 90
OVRMSGF command 331
*OWNER option 26

P
packed decimal items 268
paper positioning 233
parameters of CRTCLBP40 command 18
See also Create COBOL Program
(CRTCLBP40) command
parameters, describing in the called program 280
partial key, referring to 242
parts of a COBOL program
See program structure
parts of a program 9
passing addresses between programs 303
passing data 279
in groups 281
passing data item and its length 280
*PATCH option 21
PERFORM statement 312, 336
performance considerations 268
I/O operations 102
permanent segment 309
*PGM option 18
PGM parameter for CRTCLBP40 command 18
*PGMID option 18
phrases
ADVANCING 233
ADVANCING PAGE 336
AT END 187, 190
CORRESPONDING 20
END-OF-PAGE 336
END-REWRITE 192
END-WRITE 199
FOOTING 336
FORMAT 181, 186, 189, 191
GIVING 335, 336
INDICATORS 145, 181
INTO 186, 335
INVALID KEY 190, 192, 199
NEXT MODIFIED 189
NO DATA 186
NO REWIND 336
NOT AT END 187, 190
NOT INVALID KEY 190, 192, 199
REEL/UNIT 336
RELATIVE KEY 335
REMAINDER 336
ROLLING 195
STARTING 194
SUBFILE 182
TERMINAL 182, 187, 190, 192, 193, 198
USING 335
PICTURE clause 267, 342
and performance 268
defining with LIKE clause 258
PICTURE definitions 116
PIP (program initialization parameters) data
area 306
description 306
pointer alignment, definition 283
pointer data items
definition 282
elementary items 287
pointers
aligning on boundaries
01-level items 284
77-level items 284
portability considerations

See segmentation

position of PROCESS statement 32

preface xi

prestart job 306

previous release, compiling for 31

*PRINT option 21

printer file, default 24

printing

based on indicators 234

editing field values 234

in overflow area 234

maintaining print formats 234

multiple lines 234

output from job with WRITE statement 233

paging 233

paper positioning 233

spacing 233

procedure branching statements 348

Procedure Division

and DBCS characters 343

Procedure Division (continued)

and transaction files 176

description 10

database 397

changes from ANSI 74 COBOL 335

descriptor 10

description 311

segmentation 311

specifying debugging in 316

subdivisions in 309

USE FOR DEBUGGING declarative 316

using SET statement to specify address 287

PROCESS statement 337

allowable options for 32

compiler options specified in 32

compiler output 37

compiler considerations

blocking output records 102

commitment control considerations 94

DATABASE files 241

device dependencies 89

DISK files 241

file and record locking 93

overriding program-specified files 92

overview 255

processing methods for types DISK and

DATABASE 241

program-described and externally described

files 104

spooling 91

unblocking input records 102

COPY statement, using with 36, 37

description 32

format of 32

options 36

position of statement 32

rules for 32

scope of options with CRTCBLPGM

command 36

specifying compiler options 40

techniques

file processing 351

indexed file creation 356

indexed file updating 357

relative file creation 361

relative file retrieval 365

relative file updating 363

sequential file creation 351

sequential file updating and extension 353

using to specify compiler options 32

processing methods for DATABASE files 241

processing methods for DISK files 241
processing of I/O verbs
  since Version 1, Release 3 80
program control
  returning 274
  transferring 273
program-described files
  considerations for using 105
  description 104
  externally described by DDS with Create File commands 104
  TRANSACTION files 139
program global table (PGT), definition 371
program initialization parameters (PIP) data area
  See PIP data area
program listings, DBCS characters in 349
program loops 271
program-name 18
program object
  compiler options, specifying 21
  optimizing, specifying at compile-time 22
  output from compiler 15
  specifying authority to 27
  subscript range checking 21
program parts 9
program patch area 21
program segments 309
program size 22
program stack, definition 273
program structure
  Data Division 175
  Data Division map 44
  data field 116
  Environment Division 171
  example 9, 10
  format (record) level 115
  Identification Division 10
  indicator 116, 117
  level of language support 325
  Procedure Division 176
  required and optional divisions 10
  skeleton program 9
program syntax, debugging line 321
program template 21
program termination
  (continued)
  with the CANCEL statement 336
program variables
  changing 63
  pointers 63
program-defined key fields 246
programming considerations 255
programming notes
  number of entries in Object Definition Table (ODT) 15, 22
prompts, using SEU
  See source entry utility
  *PRTCORR option 20
  example listing 256
PRTFILE parameter for CRTCBLPGM command 24
*PRV option 26, 31
punctuation 2
purpose of this manual xi
Q
QCMDEXC, using in a program 28, 255
QLBLMSG compile-time message file 331
QLBLMSGE run-time message file 331
QLBLSRC (default source file) 10, 18
QLRCHGCM API 70
QLRRRTVCE API 70
QLRSETCE API 53, 70
QLRLMAIN
  MGTFUNC 269
QLSYSPRT (default printer file) option 24
quadruple spacing 38
*QUOTE option 20
QUOTE, figurative constant, value of 20
R
random processing, definition 365
*RANGE option 21
  reference modification 262
READ statement 344
  changes in the use of ANSI 74 COBOL 335
  description 182
  format, nonsubfile 186—187
  format, subfile 189—190
  indicators 145, 146
  processing facilities 181—182
  FORMAT phrase 181
READ WITH NO LOCK  93, 95

record boundary  23

record format
  composition for display device  140
  DDS for subfiles  159, 161
  example, record format specification  105, 107, 110
  fields  140
  indicators  142
  specification, use of DDS keywords in  107
  subfiles  157

RECORD KEY clause  129
  EXTERNALLY-DESCRIBED-KEY  129

record keys  129

RECORD KEYS, valid  242

record length of source file  10

records
  blocking output  102
  containing pointers  286
  locking
    and failed I/O  94
    and performance  270
    by COBOL  93
    updating database records  93
  preserving sequence of  250
  reducing mismatches  281
  unblocking input  102

recovery  82
  example  84
  procedure in program  83
    with multiple acquired devices  83
    with one acquired device  83
  transaction files  83
  with commitment control  82

recursive call, definition  273

REDEFINES Clause  341
  for ALL-FORMATS or I/O phrases  119

redefinition of formats  119

redefinition, group level name  116

redirecting files  90, 92

REEL/UNIT phrase  336

reference modification
  and *RANGE option  22
  and INSPECT statement  264
  calculating offset  289
  description  262
  left-justification  264
  retrieving time value  262

reference numbers  19, 43, 49

REFERENCES field  48

references to other manuals xi

referring to a partial key  242

register  319

reinitialization, avoiding  119

related printed information

relative files
  and performance  270
  creating  351, 361
  definition  249
  for OPEN OUTPUT  270
  in COBOL  249
  initializing for output  251
  retrieval of  351, 365
  sequential access  23
  updating  351, 363

relative I-O module  324

RELATIVE KEY clause  173

RELATIVE KEY phrase  335

relative key, definition  158

release-level option  26, 31

RELEASE statement  348

releasing a record read for update  93

REMAINDER phrase  336

remote systems, communications between  139, 306

RENAME keyword  124

RENAMEs clause  342

REPLACE parameter for CRTCBLGPM command  26

REPLACING, in format 2 COPY  127

reply modes  52

report writer module  325

required
  clauses  3
  divisions  10
  items, in syntax  3

reserved word list, and extended
  ACCEPT/DISPLAY  23
  reserved word, -DDS added to  123, 127

responding to messages in an interactive environment  329

restrictions  89

return codes  75

return of control from called program  274

RETURN statement  348

REUSEDLT option
  See reusing deleted records
reusing deleted records
  indexed files 242
  relative files 249
  sequential files 250
Revoke Object Authority (RVKOBJAUT)
  command 27
REWRITE statement
  and DBCS 344
  description 191
  for program-described transaction files 191
  for TRANSACTION file 191
  format 191, 192
  indicators 145, 146
  processing facilities 181, 182
ROLLBACK statement 95
  boundary 95
ROLLING phrase 195
run time
  common errors 56
  debugging 67, 314
  debugging switch 313
  error handling, de-edit 267
  messages 328
    and standard error handling 69
  monitoring exceptions 16
  program termination 52
  redirecting files 90
  subscript range checks, specifying 21
  switch 67, 314, 315
run unit
  definition 36, 273
  examples
    multiples, running consecutively 276
    single run unit 274
    with a shared program 277, 278
running COBOL/400 programs
  description 51
  system reply list and reply modes 52
RVKOBJAUT (Revoke Object Authority)
  command 27

S
S in PICTURE clause 268
SAA Common Programming Interface (CPI)
  support 325
SAA CPI (Common Programming Interface)
  support 325
SAA data types
  SAA flagging 47, 333
  SAAFLAG parameter for CRTCBLPGM
    command 25, 35
  screens
    See displays
  SEARCH statement 348
  searching DBCS characters in a table 348
  *SECLVL option 20
  SECTION field 45
  security
    maintaining while testing 55
    specifying authority to object program 27
  *SEG1 option 25
  *SEG2 option 25
  SEGMENT-LIMIT clause 310
  segment-numbers 309—311
  segmentation 268, 309—312, 325, 368
  segmented program 309
SELECT statement,
  EXTERNALLY-DESCRIBED-KEY 121
  separate indicator area (SI) attribute 143
  sequence
    combining numbers 20
    errors, checking for 19
    number 10
    of records, preserving 250
    sequence error indicator (S) 43
  *SEQUENCE option 19
  sequential access mode 23, 173, 187, 189, 249, 251
sequential files
  creation 249, 351
  definition 249
  in COBOL 249
  updating and extension 351, 353
  sequential I-O module 324
  service marks x
SET statement 346
SEU (source entry utility)
  browsing a compiler listing 39
  editing source programs 6, 9, 11
  entering source programs 6, 9, 11
  errors
    coding errors 56
    common errors 56
    detected by compiler 56
    listing 48
    messages at run time 328
    formats, using 11
    prompts and formats 11
SEU (source entry utility) (continued)
  Start Source Entry Utility (STRSEU) command 9
  syntax-checking 11—13, 327
  TYPE parameter 9
severity-level 19, 26
severity level of messages 19, 24, 330
shared files 93
shared-for-read 93
shared-for-read lock state 93
shared-for-update 93
shared-no-update 93
shared ODP (open data path) 94
shared records 93
shift-in character, definition 338
shift-out character, definition 338
SIGN clause, defining with LIKE clause 258
sign in PICTURE clause, and performance 268
sign test, de-editing 267
single device files 162
size limits, internal 15
skeleton program 9
SKIP statement 38
SKIP1 statement 38
SKIP2 statement 38
SKIP3 statement 38
slash (/) 12, 38
SORT statement 312, 348, 368
sort-merge module 324
SORT/MERGE statement 348
Source Entry Utility
  See SEU
source file
  default 10
  fields 10
  logical structure, specifying 172
  program, suppressing listing 41
  record length 10
source file format
  description 10
  program segments 309
  record length 10
source-file-member-name 19
source-file-name 18
source language debugging 313
source listing, example 41
source member type
  CICSCBL 13
  CICSSQLCBL 13
  compiling 16
source member type (continued)
  specifying 9, 11
  SQLCBL 13
  syntax-checking 11, 13
SOURCE NAME field 44
*SOURCE option 19, 37
source program
  compiling 15, 313
  counting verbs used 19
  DEBUG-ITEM special register 319
documenting lines 321
definition 2
editing source programs
  See SEU (source entry utility)
entering source programs
  See SEU (source entry utility)
listing 19, 41
WITH DEBUGGING MODE switch and compilation 313
source text manipulation module 324
spooling 91, 92
SQL (Structured Query Language) statements 12
SQLCBL member type 13
*SRCDBG option 20
SRCFILE parameter for CRTCBLPGM command 18
SRCMBR parameter for CRTCBLPGM command 18
*SRCMBRTXT option 19
SST keyword 126
stacks, shown in syntax 3
standard error handling 69, 76
standard, ANSI X3.23-1985 xiii
Start COBOL Debug (STRCBLDBG) command 314, 316
Start Debug (STRDBG) command 55
Start Source Entry Utility (STRSEU) command 9
START statement 242, 344
STARTING phrase 194
starting the compiler 194
Index 409
statement length, maximum 11, 12
statement number (STMT) field 44, 49
statement number, compiler-generated (STMT) 43

statements
ACCEPT 103, 177, 343
ACQUIRE 178
ALTER 312
arithmetic, in DBCS processing 345
breakpoints 57
CALL 312
CANCEL 336
CLOSE 179, 336
COMMIT 95
compiler output 37
COPY 104, 114, 335, 348
DISPLAY 344
DIVIDE 336
DROP 179
EJECT 38
in syntax diagrams 3
INSPECT 345
MERGE 312, 335, 348, 368
MOVE 319, 346
OPEN 180
PERFORM 312, 336
PROCESS 32, 337
READ 182, 335, 344
RELEASE 348
RETURN 335, 348
REWRITE 191, 344
ROLLBACK 95
SEARCH 348
SET 346
SKIP 38
SORT 312, 368
START 344
START, generic 242
STOP 348
STRING 346
UNSTRING 346
USE 199
WRITE 193, 336, 345
*STDERR option 22
*STDNZ option 23
STOP RUN statement 274, 306
STOP statement 348
storage optimization
See segmentation
storage, initialization of 279
storage, using less 22
STRCBLDBG (Start COBOL Debug) command 314, 316
STRDBG (Start Debug) command 55
STRING statement 346
STRSEU (Start Source Entry Utility) command 9
structure, program
See program structure
Structured Query Language (SQL) statements 12
subfield contents, DEBUG-ITEM special register 319
subfiles 156—158, 182
subprogram 36, 273
linkage 281
subscript range checking, specifying 21
subscript ranges 21
subscripting 265
substitution character (X’3F’) in data 137
suffix -DDS
added to key field name 123, 125
added to reserved word 123, 127
summary of changes
changes made in Version 2 Release 1.1
changes made in Version 2 Release 2
support for ANSI X3.23-1985 standard 323
suppressing source listing 41
suppression of messages 330
switch, run-time 67, 314, 315
symbols used in syntax 3
*SYNC option 22
syntax
arrows 3
checking, in SEU 11, 12, 39
checking, unit of 11
debugging lines 321
diagrams, using 3
keywords in 2
notation 2
of CRTCBLPGM command 29
optional items 3
punctuation 2
required and optional clauses 3
required items 3
stacks 3
symbols 3
system override considerations 92
system reply list 52
T

transaction files (continued)
display management 140
Environment Division considerations 171
externally described 139
file control entry and Environment Division 171
file description entry and Environment Division 175
FILE STATUS clause 173
file status, setting of 71
major return code 71
minor return code 71
ORGANIZATION clause 172
organization of 172
Procedure Division considerations 176
processing externally described 142
program-described
RELATIVE KEY clause 173
return codes 71
sample programs, work station 200
transferring control to another program 273
transferring program control 273
triple spacing 38

U

UFCB (user file control block) 71
unattended mode, running the program 331
unblocking input records 102
underscores, removed from end of field name 113
underscores, translated to hyphens 113
*UNDSPCHR option 25
unit of syntax checking 11
*UNREF option 22
unreferenced data items 22
unreferenced identifiers 15
UNSTRING statement 346
updating
and extension of sequential files 351, 353
indexed files 351, 357
relative files 351, 363
sequential files 353
UPSI (user program status indicator) switch
USAGE clause
defining with LIKE clause 258
numeric 116
USAGE IS POINTER 282
with transaction files 144
USE FOR DEBUGGING declarative 316, 317
in the Procedure Division 316
using the procedures 317
*USE option 27
USE procedure
role since Version 1, Release 3 80
USE statement
coded examples 353, 354
description 199
EXCEPTION/ERROR for TRANSACTION file 199
format 199
user file control block (UFCB) 71
*USER option 26
user profile 26
user program status indicator (UPSI) switch
user spaces
accessing using APIs 291
using a subfile for display 156—158
using double-byte characters 337
using less storage 22
USING phrase 335
using REPLACING in format 2 COPY statement 127
using the COBOL/400 language
See COBOL/400 language
USRPRF parameter for CRTCBPGM command 26

V
V2R1M0 option 31
V2R1M1 option 31
V2R2M0 option 31
valid RECORD KEYS 242
validity checking 140
VALUE clause 342
VALUE IS NULL 304
value of figurative constant QUOTE 20
*VARCHAR option 23
variable-length fields 131
defining 131
element of example of 131, 134
length of, example of 132
maximum length of 131
restrictions 131
variables, changing values while testing 63
*VBSUM option 19, 37
verbs usage by count listing 43

W
where DBCS characters can be used 340
WITH DEBUGGING MODE switch and compilation 313
work stations
communications between 139
sample programs
order inquiry 206
payment update 217
transaction inquiry 200
validity checking 140
Writing-Storage section
defining identifiers 258
WRITE statement
and DBCS 345
changes from ANSI 74 COBOL 336
description 193
for program-described transaction files 193
for TRANSACTION file 193
format, nonsubfile 193—195
format, subfile 198—199
indicators 145, 146
processing facilities 181—182

X
X'3F' (substitution character) in data 137
*XREF option 19, 21, 37